НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ім. ІГОРЯ СІКОРСЬКОГО»

Факультет Інформатики та обчислювальної техніки

Обчислювальної техніки

До захисту допущено:

Завідувач кафедри

_____ Сергій Стіренко

“___”_____2020р.

Дипломна проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Комп’ютерні системи та мережі»

спеціальності 123 «Комп’ютерна інженерія»

на тему: «Додаток побудови маршрутів»

Виконав: студент 4 курсу, групи ІО-63
Іщенко Богдан Андрійович _____ (підпис)

Керівник:
ст. викладач Алещенко Олексій Вадимович _____ (підпис)

Консультант з нормо-контроль:
проф. д.т.н. Сімоненко Валерій Павлович _____ (підпис)

Рецензент:
Радченко Костянтин Олександрович _____ (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ - 2020 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМ. ІГОРЯ СІКОРСЬКОГО”

Факультет Інформатики та обчислювальної техніки

Обчислювальної техніки

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 123 «Комп’ютерна інженерія»

Освітньо-професійна програма «Комп’ютерні системи та мережі»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Сергій Стіренко
“___”_____2020р.

ЗАВДАННЯ

на бакалаврський дипломний проект студента

Іщенка Богдана Андрійовича

1. Тема проекту (роботи) _____*“Додаток побудови маршрутів*”
Керівник проекту (роботи) _____*ст. викладач Алещенко Олексій Вадимович*
затверджені наказом по університету від _____*«07» травня 2020 р. № 1081-с*
2. Термін здачі студентом закінченого проекту (роботи) _____*5 травня 2020*р.
3. Вихідні дані до проекту (роботи) *технічна документація. теоретичні та дані, прототип робочого додатку*
4. Зміст розрахунково-пояснювальної записки *Опис предметної області, дослідження аналогів, дослідження засобів написання додатку, інструкція з використання додатку.*
Перелік графічного матеріалу (з точним позначенням обов’язкових креслень)
структурна схема, схема взаємодії, функціональна схема
5. Консультанта проекту (роботи), з вказівкою розділів роботи, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання	Завдання прийняв
Нормо контроль	Сімоненко В. П.		

7. Дата видачі завдання _____

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Найменування етапів дипломного проекту (роботи)	Строк виконання етапів проекту(роботи)	Примітки
1.	*Вивчення літератури*	*20.03.2020*	
2.	*Складання та узгодження технічного завдання*	*01.04.2020*	
3.	*Створення модулів системи що розробляється*	*10.04.2020*	
4.	*Тестування окремих модулів системи*	*20.04.2020*	
5.	*Доопрацювання, налагодження і виправлення помилок*	*01.05.2020*	
6.	*Оформлення документації дипломної роботи*	*1.05.2020*	
8.	*Передзахист*	*26.05. 2019*	
9.	*Захист*		

Студент-дипломник _____(Іщенко Богдан)
(підпис)

Керівник роботи _____(Алещенко Олексій)
(підпис)

Анотація

Робота присвячена розробці навігаційної програми для смартфонів на базі операційної системи Android. Було розроблено зручний та інтуїтивний інтерфейс, що дозволяє швидко будувати потрібний маршрут, враховуючи різні параметри такі як вартість поїздки та кількість викидів вуглецевого газу в ході поїздки. Користувач може редагувати частини маршруту, змінюючи тип подорожі а більш зручний.

Для написання додатку була використана мова програмування Java 7.0. В додатку виклики PI методів здійснюються фреймворком Retrofit 2.0. Для забезпечення асинхронної роботи в використано фреймворк RxJava 2. В якості основного остачальника карт виступає Google Maps SDK. Екран нафігації працює на базі Mapbox SDK.

Аннотация

Работа посвящена разработке навигационной программы для смартфонов на базе операционной системы Android. Был разработан удобный и интуитивный интерфейс, позволяющий быстро планировать нужный маршрут, учитывая стоимость поездки и количество выбросов углекислого газа в ходе поездки. Пользователь может редактировать сегменты маршрута, изменяя тип передвижения на более удобный.

Для написания приложения был использован язык программирования Java 7.0. Для работы с API было использовано фреймворк Retrofit 2.0. За асинхронность работы в проекте отвечает фреймворк RxJava 2. Как основной поставщик карт выступает Google Maps SDK. Для работы навигатора был было использовано Mapbox SDK.

Abstract

The work is devoted to the development of a navigation program for smartphones based on the Android operating system. A user-friendly and intuitive interface has been developed, allowing you to quickly plan your route, taking into account the cost of the trip and the amount of carbon emissions during the trip. The

user can edit the route segments, changing the type of movement to a more convenient one.

Java 7.0 programming language was used to write the application. The Retrofit 2.0 framework was used to work with the API. The RxJava 2 framework is responsible for the asynchrony of work in the project. The Google Maps SDK acts as the main provider of maps. The Mapbox SDK was used for the navigator.

ОПИС АЛЬБОМУ

ВІДОМІСТЬ ДИПЛОМНОГО ПРОЕКТУ

№ з/п	Формат	Позначення	Найменування	Кількість листів	Примітка
1	А4		Завдання на дипломний проект	2	
2	А4	ІАЛЦ.467100.001 ВП	Відомість проекту	1	
3	А4	ІАЛЦ.467100.002 ТЗ	Технічне завдання	3	
4	А4	ІАЛЦ.467100.003 ПЗ	Пояснювальна записка	65	
5	А4	ІАЛЦ.467100.004 Д1	Структурна схема	1	
6	А4	ІАЛЦ.467100.005 Д2	Схема взаємодії	1	
7	А4	ІАЛЦ.467100.006 Д3	Функціональна схема	1	
8	А4	ІАЛЦ.467100.007 Д4	Лістинг	12	

					ІАЛЦ.467100.001 ВП		
Змн.	*Арк.*	*№ докум.*	*Підпис*	*Дата*			
Розроб.		*Іщенко Б.А.*			*Відомість дипломного проекту Додаток побудови маршрутів*	*Літ.*	*Арк.*	*Акрушів*
Перевір.		*Алещенко О.В.*					*1*	*1*
						КПІ ім. Ігоря Сікорського ФІОТ ІО-63		
Н. Контр.		*Симоненко В.П.*						
Затверд.		*Стиренко С.Г.*						

ТЕХНІЧНЕ ЗАВДАННЯ

ЗМІСТ

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку програми для смартфона на базі операційної системи Android.

Область застосування: Повсякденне планування маршрутів, а також навігаційя по цих маршрутах.

2. ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на виконання роботи кваліфікаційно-освітнього рівня «бакалавр комп'ютерної інженерії», затверджене кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут».

3. МЕТА І ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою даного проекту є розробка програми для зручної повсякденної навігації по місту і за ним.

4. ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки є науково-технічна література з теорії і практики програмування, бакалаврські роботи інших студентів, публікації в Інтернеті з даних питань.

5. ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до вихідного продукту

- Простий і інтуїтивно-зрозумілий інтерфейс системи.
- Можливість планувати маршрут на майбутню дату
- Нагадування про заплановану поїздку
- Можливість змінювати сегменти маршруту (тип транспорту при його зміні)

5.2. Вимоги до програмного забезпечення

- Операційна система Android (API 22-29).

6. ЕТАПИ РОЗРОБКИ

	Дата
Вивчення літератури	20.03.2020
Складання і узгодження технічного завдання	01.04.2020
Створення модулів системи, що розробляється	10.04.2020
Тестування окремих модулів системи	20.04.2020
Допрацювання, налагодження і виправлення помилок	01.05.2020
Оформлення документації дипломної роботи	10.05.2020

ПОЯСНЮВАЛЬНА ЗАПИСКА

ЗМІСТ

ВСТУП

Головним завданням цієї роботи є створення зручного навігаційного додатку для смартфона. Багато людей стикається з проблемою побудови оптимального маршруту подорожі. Раніше для цього використовували паперові карти, згодом їх перевели в електронну форму, яку зараз перенесли в більшість електронних девайсів, щоб ці карти були завжди під рукою. Кожен розробник у даній сфері представляє користувачу свій варіант розв'язання проблеми з навігацією на певній території. Отож ця робота не буде першою у сфері навігаційних застосунків, але в ній буде розроблено функціональність, якої бракує іншим, подібним розробкам.

Розроблений застосунок отримав зручний та інтуїтивний інтерфейс, щоб задовольнити одне із завдань роботи: користувачу має бути зрозуміло як користуватися застосунком відразу після першого запуску, він має без проблем розуміти як прокласти маршрут будь-якої складності: від поїздки на роботу до планування подорожі в іншу країну.

Другим завданням роботи є створення інтерфейсу для планування поїздок в майбутньому. Для цього в додатку поєднано календар та карти. Користувач може створити подію на певну дату і час, та маршрут до місця цієї події. У ході її створення додаток запропонує користувачу маршрути з урахуванням розкладу громадського транспорту та прогнозованої транспортної ситуації на запланований період часу. Перед стартом події (з урахуванням часу подорожі) користувач отримає нагадування про неї та зможе в один клік перейти до навігаційної частини застосунку.

Третім завданням роботи є організація можливості змінити транспорт при пересадці. Наприклад, якщо користувачу було запропоновано маршрут, в який входить декілька різних транспортних засобів, додаток допоможе обрати на який саме транспортний засіб користувач буде пересаджуватися (метро, автобус, таксі).

Відповідно до вище перерахованих завдань була сформована тема роботи.

РОЗДІЛ 1

АНАЛІЗ НАЯВНИХ ДОДАТКІВ ДЛЯ ПОБУДОВИ МАРШРУТІВ

1.1.Опис завдання

Завдання даної роботи полягає в створенні застосунку для побудови маршрутів. Розроблений застосунок повинен мати зручний та інтуїтивний інтерфейс.

Також додаток повинен мати інтерфейс для планування поїздки в майбутньому. Користувач матиме змогу створити подію на певну дату і час, та дорогу до місця цієї події. У ході створення нової події додаток запропонує користувачу маршрути з урахуванням розкладу громадського транспорту та прогнозованої транспортної ситуації на запланований період часу. Перед стартом події (з урахуванням часу подорожі) користувач отримає нагадування про неї, та зможе в один клік перейти до навігаційної частини застосунку.

Застосунок також дозволяє редагувати вид транспорту, який користувач буде змінювати в ході подорожі.

Ще одним завданням буде дати додатку можливість автоматично визначати адресу дому та роботи користувача.

1.2.Аналіз наявних рішень

Існує безліч навігаційних додатків, і кожен спеціалізований на якомусь окремому завданні. Розглянемо 4 найбільш популярні застосунки, кожен з яких має свій напрям розвитку, свої власні недоліки та переваги, що принесе трішки різноманітності в наш аналіз.

1.2.1. Google Maps.

Google Maps [1] є найпопулярнішим застосунком у своєму роді. Програма прекрасно виконує своє завдання - планування маршрутів та навігації.

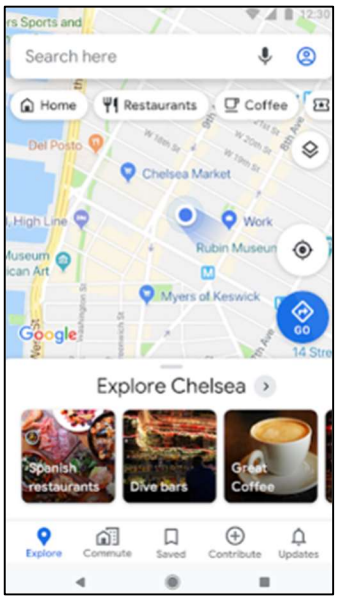


Рис. 1.1. Головний екран програми “Google Maps” [1]

Незважаючи на широкий спектр можливостей даного застосунку, головний напрям цієї програми – навігація по місту, пошук найближчих ресторанів, кафе та готелів. Додаток може показати вам робочі години закладу, його номер телефону, вебсайт, фотографії та відгуки інших користувачів про цей заклад.

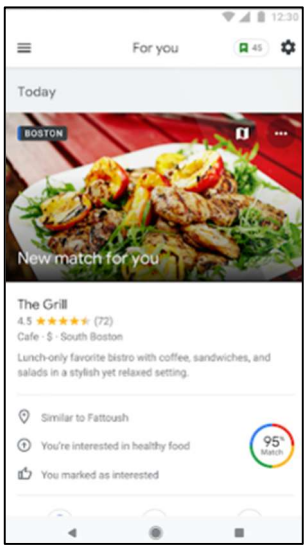


Рис. 1.2. Екран “обраного закладу” програми “Google Maps” [1]

Сильною стороною додатку є планування маршрутів в місті. Додаток запропонує вам різні маршрути громадського транспорту, серед яких ви зможете обрати найзручніший для вас, враховуючи час поїздки, кількість пересадок, ціну, тип транспорту (метро, автобус).

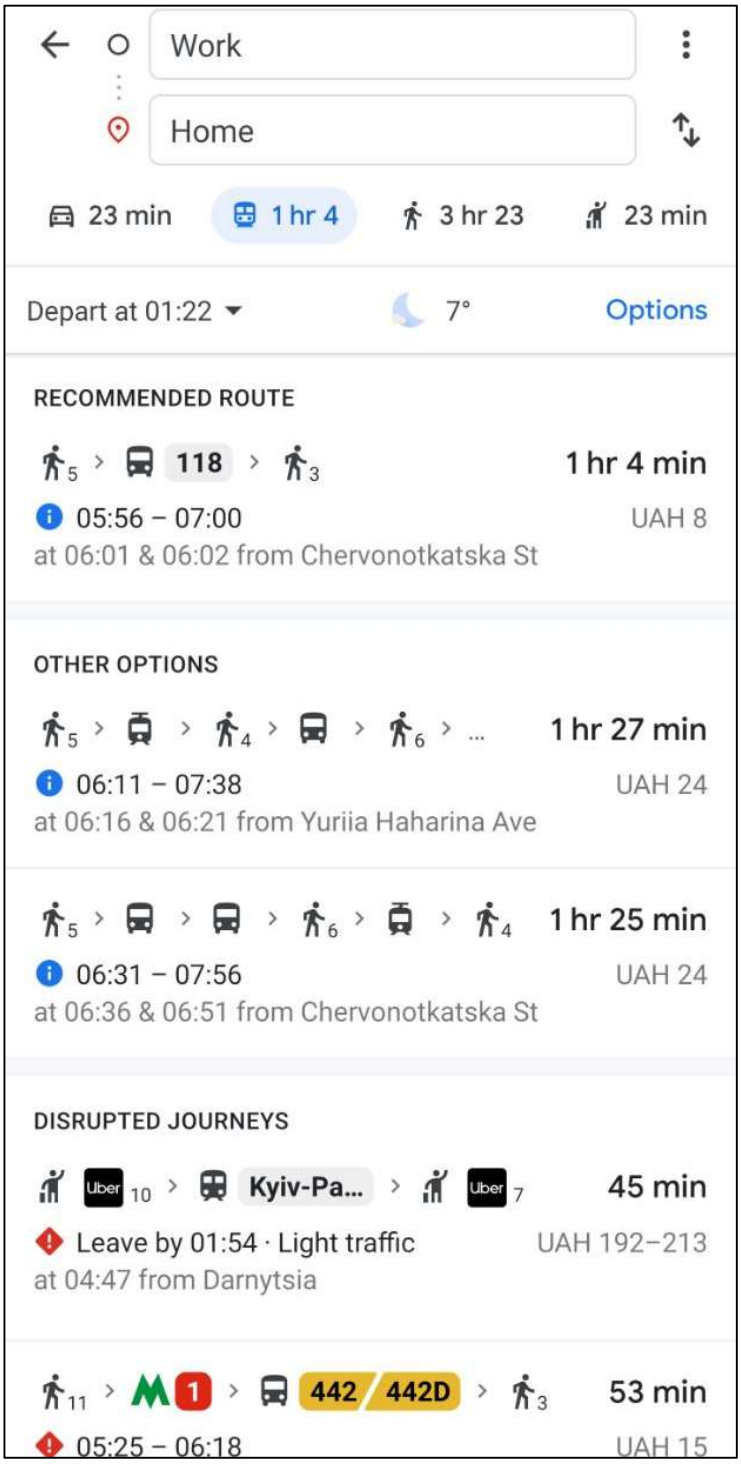


Рис. 1.3. Екран “вибору маршруту” програми “Google Maps” [1]

Яке можна бачити на рисунку 1.3, додаток надає всю необхідну інформацію для подорожі в інтуїтивно доступних місцях. Також програма має зручний навігатор для комфортного руху побудованими маршрутами. Він згодиться для пересування будь-яким видом транспорту. Всі ці аспекти роблять її одним із лідерів у своїй галузі.

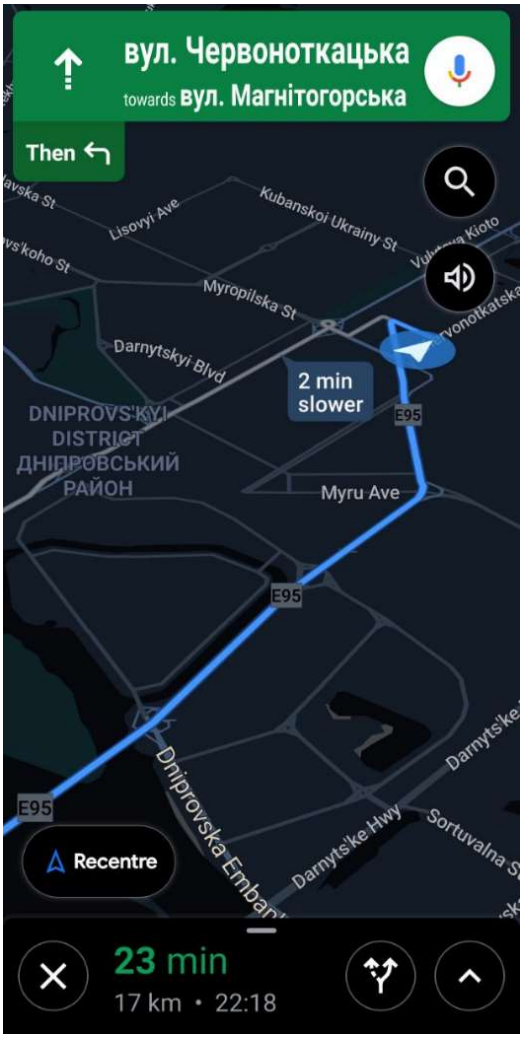


Рис. 1.4. Екран “навігації” програми “Google Maps” [1]

1.2.2. MAPS.ME

Maps.Me [2] є лідером серед навігаційних додатків з можливістю подорожі без підключення до мережі інтернет та позиціонується як навігаційний застосунок для туристів за кордоном. Користувач може завантажити карту регіону, який його цікавить, та скористатися нею за необхідності.

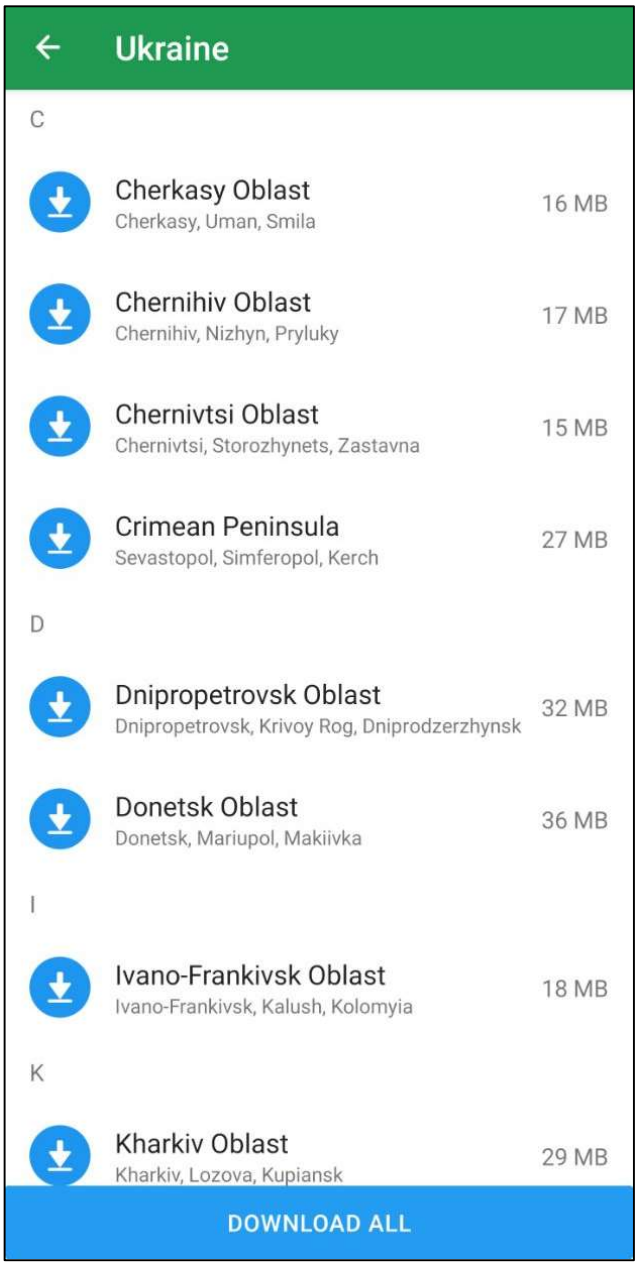


Рис. 1.5. Екран “завантаження регіонів” програми “ Maps.Me” [2]

Застосунок задля зручності туриста показує дуже деталізовану карту. Користувач може клацнути на будь-який об’єкт на карті та прочитати інформацію про нього. Наприклад, ви клацнете на музей, вам покажуть його ціну, години роботи та коротку інформацію з вікіпедії. Додаток запропонує схожі місця поруч та, якщо ви знаходитеся далеко від цього місця, запропонує замовити таксі або побудувати маршрут. Якщо під час навігації у вас зникне підключення до мережі інтернет, програма просто перейде в “offline” режим, де ви так само просто зможете дістатися пункту призначення.

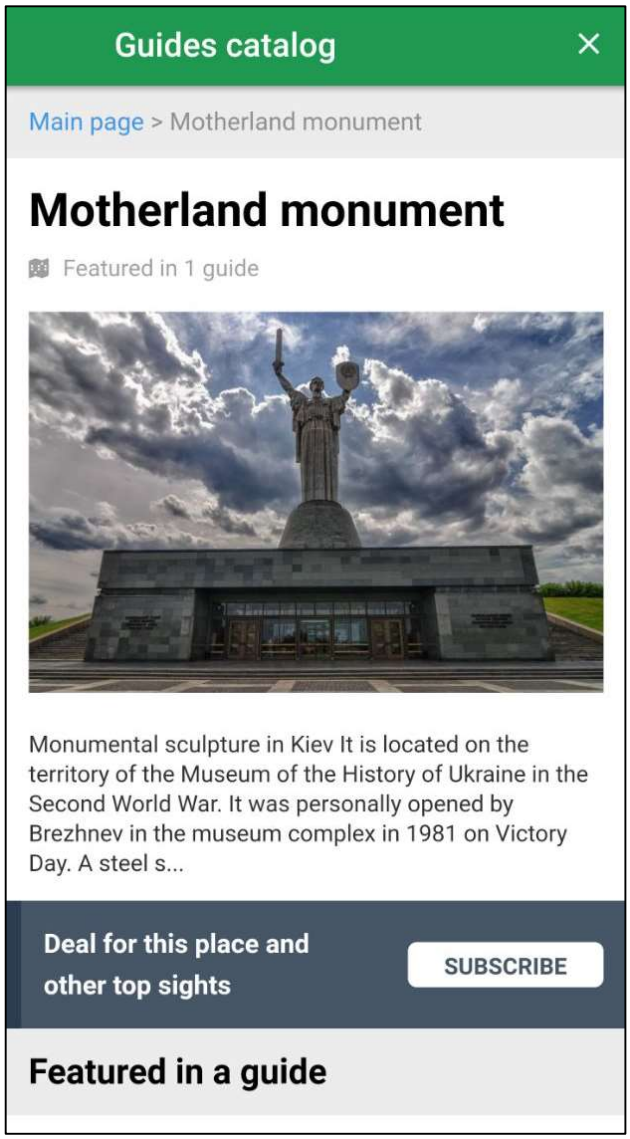


Рис. 1.6. Екран “деталей про місце” програми “ Maps.Me” [2]

Для популярних туристичних міст додаток може запропонувати онлайн тури. Користувач обирає тур, який йому найцікавіший, та переглядає його деталі (час проходження, довжина, кількість місць та список цих місць). Якщо йому тур подобається, він може заплатити символічну платню та отримати можливість зручної навігації по цьому туру.

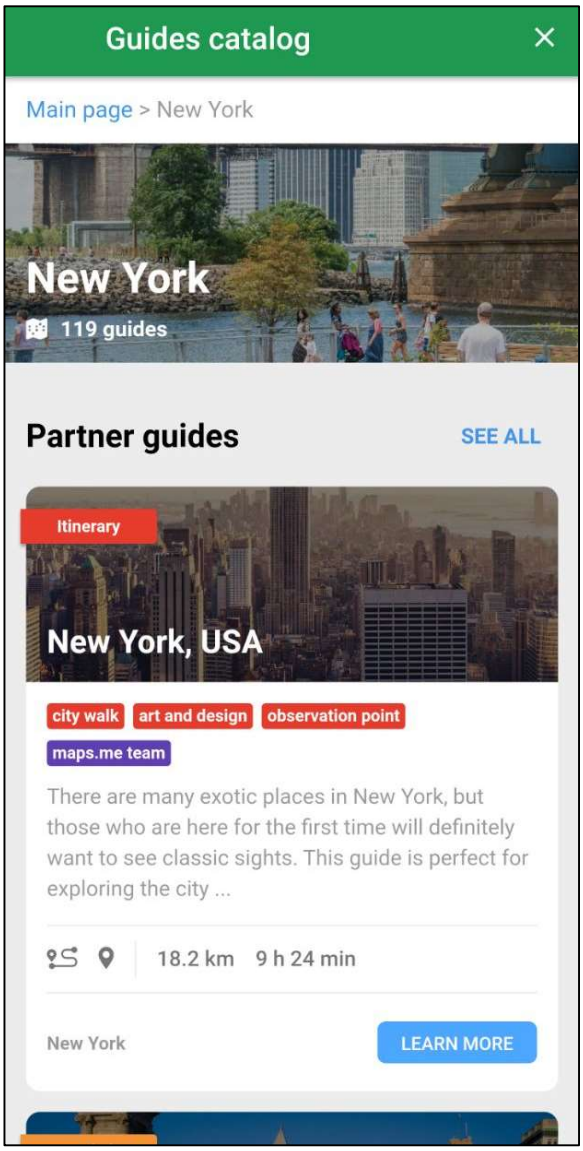


Рис. 1.7. Екран “Вибір туру” програми “ Maps.Me” [2]

1.2.3. Waze

Застосунок “Waze” [3] є лідером в галузі автомобільної навігації. Має свій унікальний, легкий для сприйняття дизайн та корисні для водіїв функції.

Робота додатку базується на місці розташування користувача та адресі, яку він ввів. Після підтвердження місця призначення програма побудує найоптимальніший маршрут (з урахуванням швидкості дорожнього трафіку) до бажаної точки на карті.

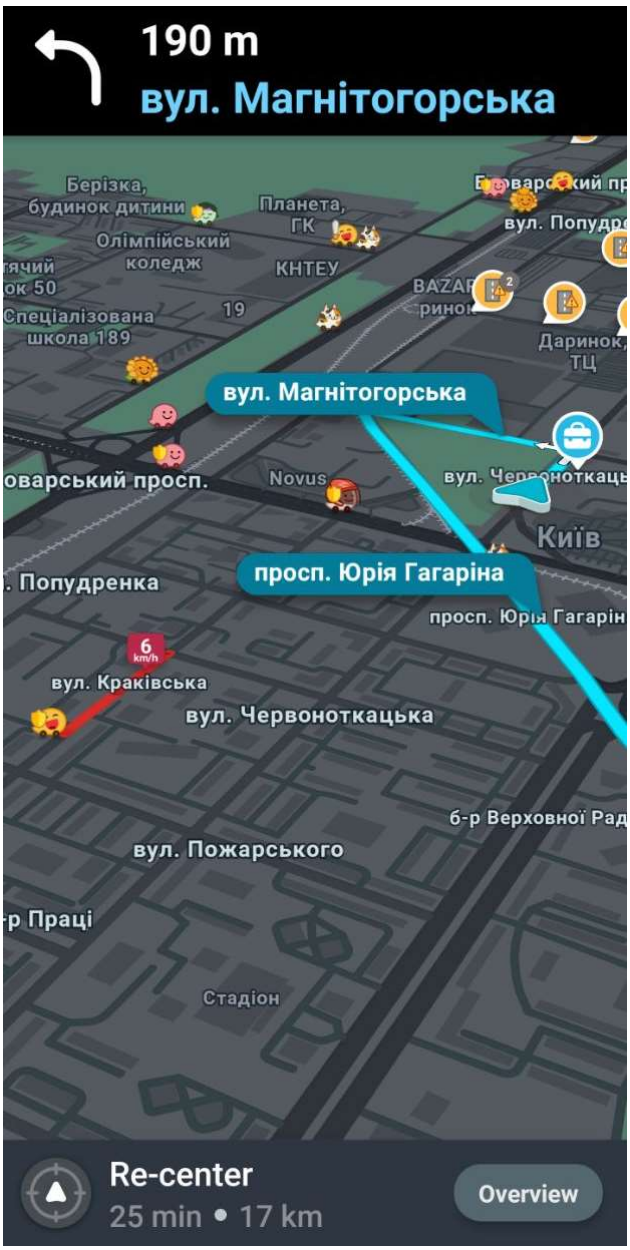


Рис. 1.8. Екран “маршруту” програми “ Waze” [3]

Програма має підказки багатьма мовами про різні дорожні ситуації. Під час навігації користувачу надається всі необхідна інформація для комфортної подорожі. Додаток повідомить користувача про:

- вибоїну попереду;
- затор (його довжину та середню швидкість);
- камеру відео фіксації швидкості дорожнього руху;
- обмеження швидкості на ділянці дороги по якій прямує водій;
- поліцейський патруль;

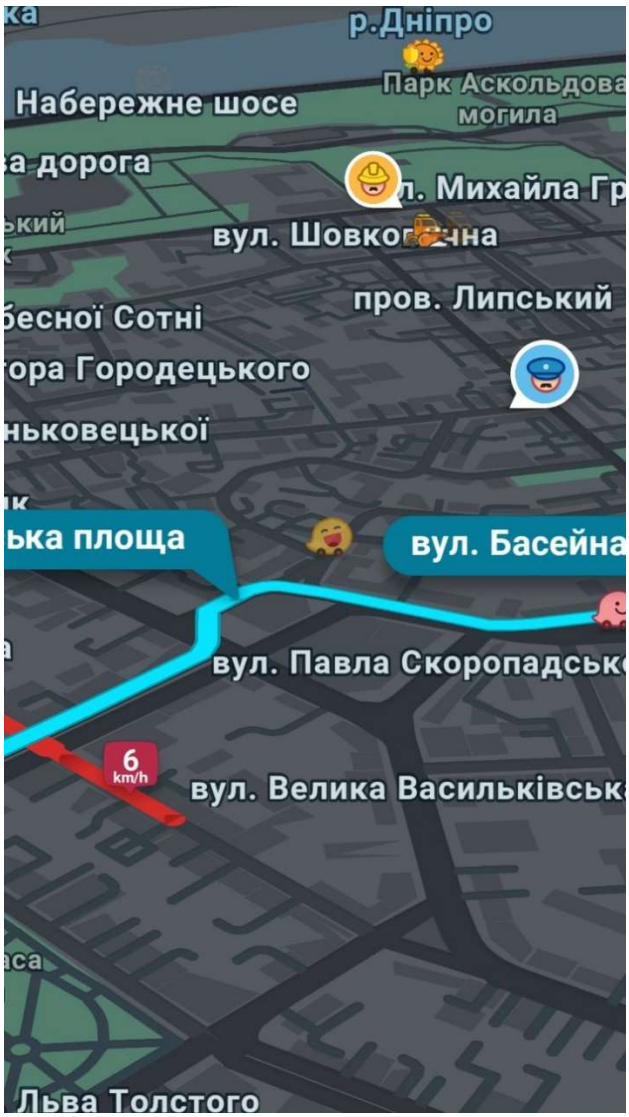


Рис. 1.9. Екран програми “Waze” з зображеним затором та поліцейським патрулем [3]

Всі ці можливості програми досягаються завдяки взаємодії всіх її користувачів.

Точність заторів вираховується завдяки тому, що застосунок постійно надсилає на сервер дані про геолокацію користувача та його швидкість. Ці дані надходять на сервер від кожного користувача, там вони обробляються та надсилаються назад користувачам у вигляді мапи заторів.

Всі дані, які показує додаток є актуальними завдяки тому, що кожен користувач має змогу підтримувати дані оновленими. Якщо користувач, скажімо, бачить яму на дорозі чи поліцейський патруль, він легко (в один клік)

може викликати меню надсилання звіту, де серед запропонованих варіантів він, в 1 клік, обирає інцидент, який з ним стався. Додаток надсилає ці дані на сервер, доповнюючи їх геолокацією користувача. Після чого ці дані з сервера потрапляють усім іншим користувачам.

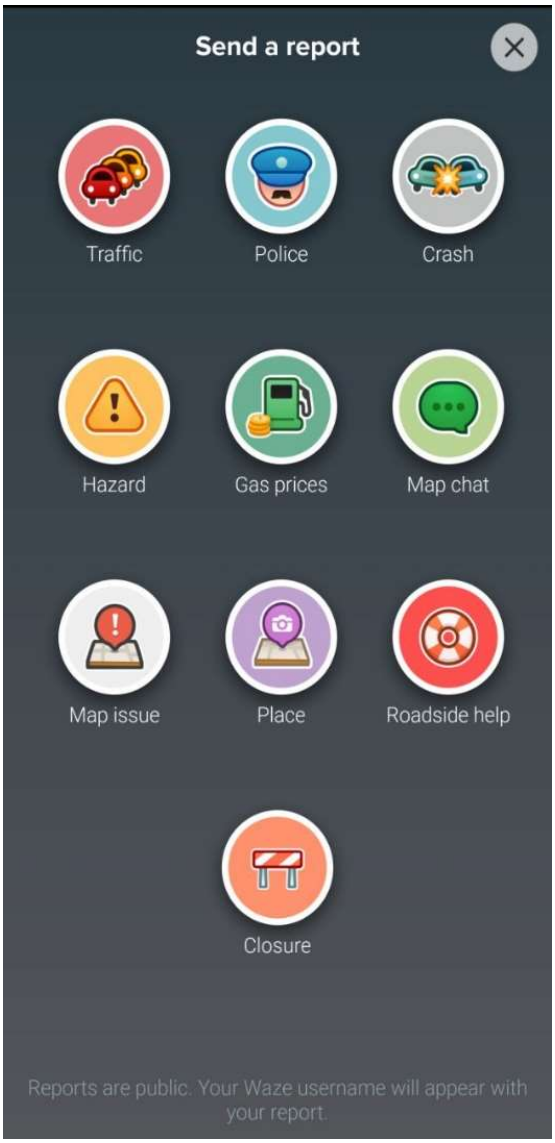


Рис. 1.10. Екран програми “ Waze” з вибором варіанту звіту [3]

ВИСНОВКИ ДО РОЗДІЛУ 1

В цьому розділі були розглянуті декілька навігаційних додатків. Кожен з них має свої переваги та недоліки, але жоден з них не має повної функціональності, яку планується запровадити в рамках цієї роботи, а саме: планування та збереження подорожі на конкретний час у майбутньому з можливістю отримати нагадування та редагування транспорту, який користувач міняє протягом подорожі.

Враховуючи вище наведений аргумент, можна вважати розробку власного навігаційного додатку актуальною, адже вона має переваги розглянутих альтернатив, враховує їх недоліки та реалізовує запропоновані вище функції.

РОЗДІЛ 2

АНАЛІЗ ЗАСОБІВ ТА ФРЕЙМВОРКІВ ВИКОРИСТАНИХ ДЛЯ РОЗРОБКИ ЗАСТОСУНКУ

2.1.Архітектура MVP

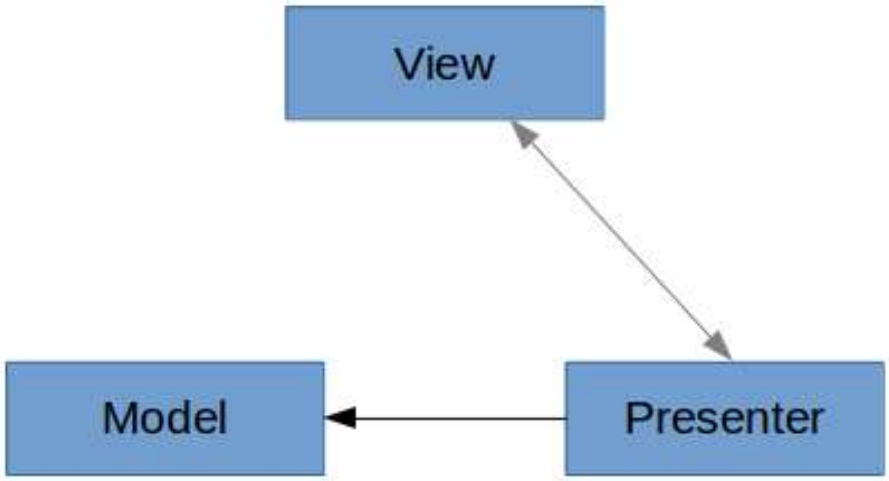


Рис. 2.1. Схема роботи архітектури MVP

Шаблон Model View Presenter [1] (MVP) створений для покращення архітектуру додатків, а саме для підвищення ефективності розуміння коду. Шаблон MVP відокремлює модель даних від представлення даних через Presenter

View – це модуль, який представляє собою користувальницький інтерфейс для представлення даних з *Model* та даних після передання команд користувача в *Presenter* та їх обробки там.

Model – це інтерфейс, який визначає модель даних, які повинен відображати користувальницький інтерфейс, модель містить відповідну бізнес-логіку.

Presenter – містить обробку подій компонентів, відповідає за отримання даних з моделі та передачу отриманих даних у *View* через перетворення формату.

Шаблон дизайну MVP зазвичай додає *Controller* як допоміжний модуль загальної програми.

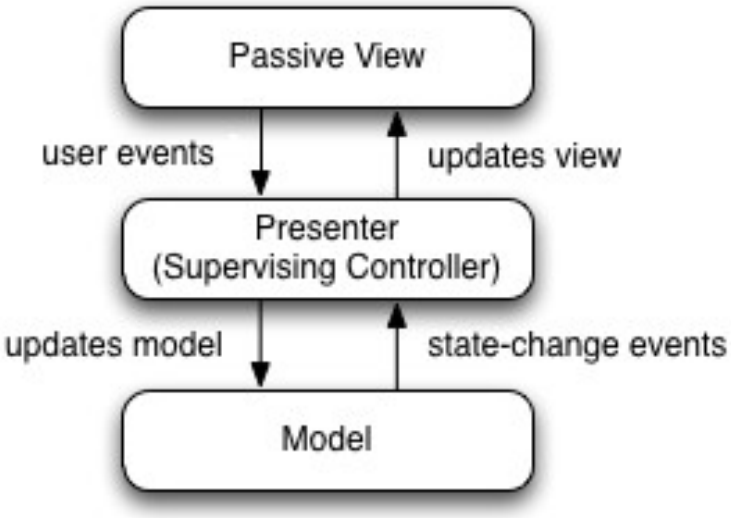


Рис. 2.2. Схема роботи архітектури MVP з візуалізацією викликів

До *Model* можна віднести такий функціонал:

- REST API.
- База даних SQLite.
- Відловлення поширених повідомлень.
- Кеш.
- Firebase.
- Retrofit.

До *View* можна віднести такий функціонал:

- Меню,
- Дозволи.
- Слухачі подій.
- Показ діалогів, спливаючих текстів,

- Робота з класами Android, такими як *View* та *Widget,*
- Старт нових *Activity***,**
- Увесь функціонал, що має відношення до класу *Context*.

До *Presenter* можна віднести такий функціонал:

- Обробка даних,
- Обробка стану екрану (прогрес, пустий, помилка, тощо),
- Керування видимістю,
- Перевірка правильності введення,
- Виклики в *View*,
- Виклики в *Model***.**

2.2. Контролер екранів

В ході роботи було розроблено модуль контролю екранів. Цей клас було створено для того, щоб полегшити рутинні перемикання фрагментів. Він має кілька методів викликами яких можна контролювати стан екрану.

Наприклад, якщо користуватися стандартними бібліотеками Android звичайне перемикання фрагменту займало б від 5 до 7 рядків коду, і складалося б з таких етапів:

- створення фрагменту,
- створення анімації,
- створення транзакції екранів,
- налаштування транзакції створеними об’єктами,
- дістати з активності менеджер фрагментів та елемент де цей фрагмент буде розміщено,
- застосувати цю транзакцію.

Фрагмент – обгортка для контенту, який бачить користувач, надає контенту життєвий цикл та реалізую логіку контенту (взаємодія користувача з додатком та додатку з користувачем)

При створенні контролера необхідно надати йому менеджер фрагментів. В конструкторі цього класу прописаний код, який ініціює стек фрагментів, ініціює корінні фрагменти та налаштовує анімацію переходів.

Створений контролер позбавляє програміста необхідності щоразу писати ці кроки, все що необхідно - це викликати метод *add()* та надати йому фрагмент. Всі інші кроки клас проходить при створенні, що також позитивно впливає на ефективність його роботи.

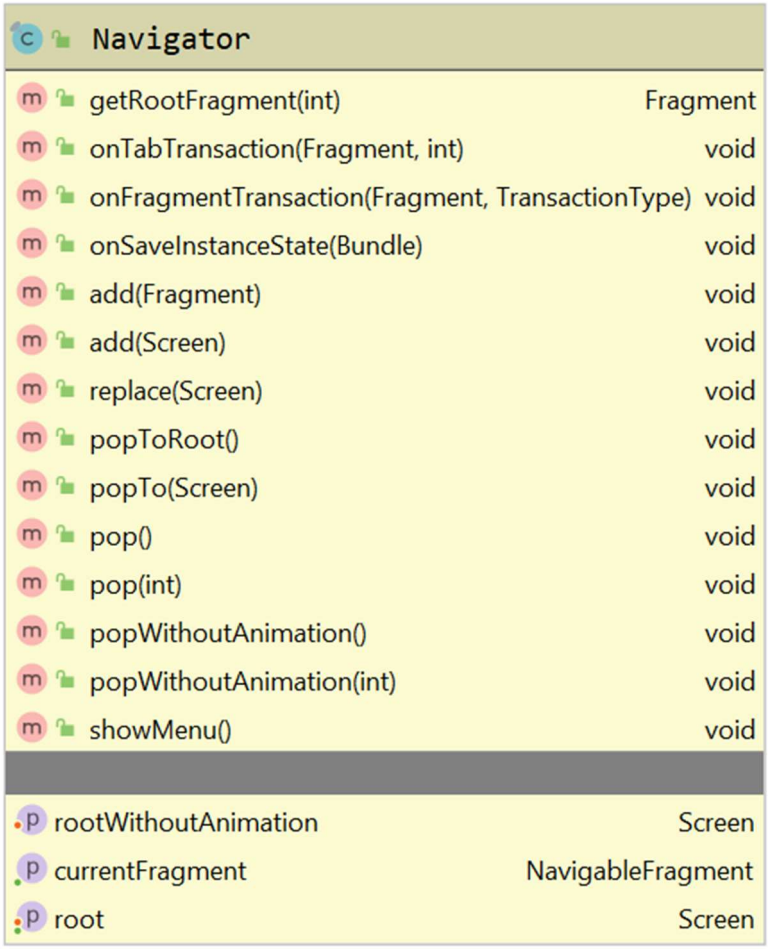


Рис. 2.3. Діаграма класу Navigator з його методами та полями

Принцип роботи контролера полягає в тому, що він розташовує фрагменти в стеку. Для керування класом написані методи аналогічні викликам до стеку: *add(), pop(), get(), replace()*.

Для зручності роботи з контролером було створено допоміжний нумерований клас *Screen*, який відповідає за життєвий цикл фрагментів. За необхідністю вони створюються та видаляються коли вони вже не потрібні.

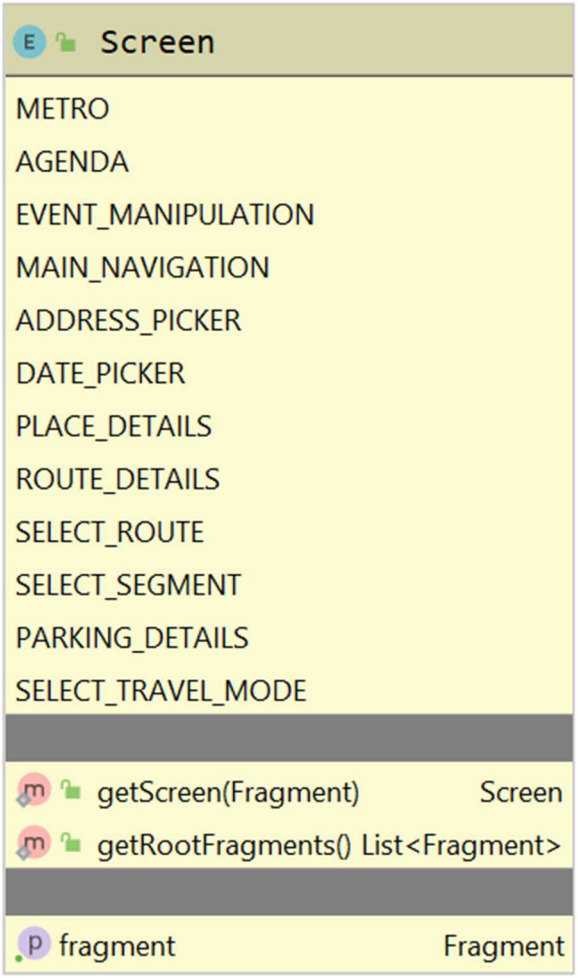


Рис. 2.4. Діаграма класу Screen з його методами, полями та варіантами

Ці 2 класи працюють в зв'язці. Програмісту достатньо просто написати виклик до контролера, указавши назву екрану з допомогою класу *Screen*, далі програма автоматично дістане створений об'єкт типу *Fragment* та виконає над ним всі необхідні дії.

```
@OnClick(R.id.address_path_input)
protected void onAddressPath() {
    navigator.add(ADDRESS_PICKER);
}
```

Рис. 2.5. Приклад додавання нового екранну

Також контролер може повертатися до конкретно вказаного екрану, якщо такий був доданий або повертатися на домашній екран.

```
@Override
public void onBackPressed() {
    navigator.popTo(SELECT_ROUTE);
}
```

Рис. 2.5. Приклад повернення до вказаного екранну

В програмі є декілька місць де після виконання певної дії потрібно повернутися на декілька екранів назад, для цього використовується такий алгоритм:

- В стеку екранів шукаємо перший з початку фрагмент що співпадає з наданим екраном;
- Знаходимо його індекс;
- Обрізаємо стек до нового значення.

```
public void popTo(Screen screen) {
    final Stack<Fragment> currentStack = fragNavController.getCurrentStack();
    final Fragment fragment;
    if (currentStack != null) {
        fragment = firstOrNull(new ArrayList<>(currentStack),
                               it -> screen.equals(Screen.getScreen(it)));
        int newSize = currentStack.size() - currentStack.indexOf(fragment) - 1;
        fragNavController.popFragments(newSize);
        getCurrentFragment().onShow();
    }
}
```

Рис. 2.5. Алгоритм повернення до певного екрану

2.3. Контролер динамічних списків

Система Android має вбудовану бібліотеку для роботи зі списками під назвою *RecyclerView*. Для створення списку з допомогою цієї бібліотеки програмісту необхідно ініціювати сам *RecyclerView*, це елемент інтерфейсу який відображає надані дані користувачу. Це можна зробити в xml файлі, або в коді фрагмента.

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/addresses_recycler_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

Рис. 2.6. Ініціація Recycler View в xml файлі

Наступним кроком є ініціація адаптера для цього списку. Він є тією частиною цієї бібліотеки, за яку її полюбили. Відповідає цей клас за створення, видалення, оновлення елементів списку. Головною його перевагою є те, що він має фіксовану кількість створених елементів. Наприклад, якщо користувач прокручує список вгору, то той елемент списку який зник в верхній частині екрану одразу з'явиться з оновленими даними внизу екрану. Таким чином досягається постійна кількість елементів, їх створюється рівно стільки, скільки потрібно, щоб заповнити сам *RecyclerView,* і не більше.

Для цього класу було створено розширення *BaseRecyclerAdapter* для кращої взаємодії з даними. Цей клас при створенні приймає в себе 2 типи даних: тип самих даних та тип елемента списку.

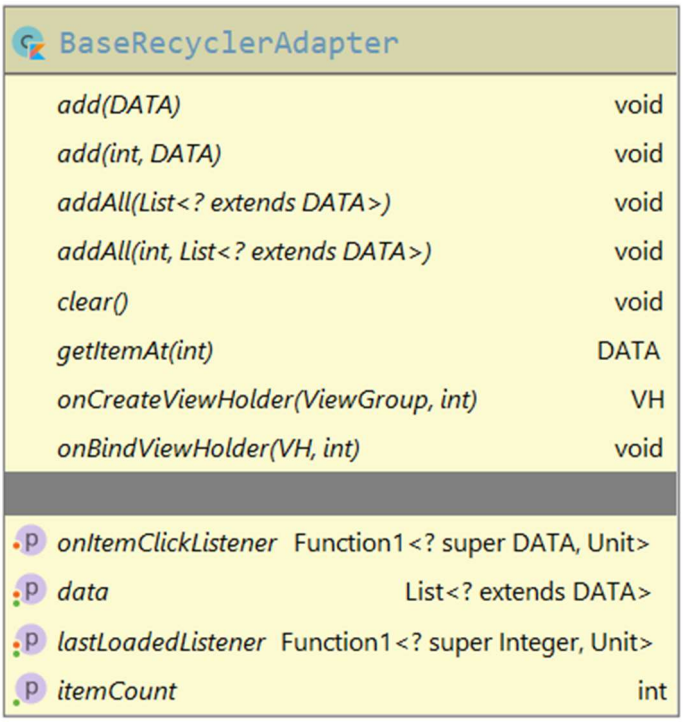


Рис. 2.6. Діаграма класу *BaseRecyclerAdapter* з його методами та полями.

В класі є два типи методів, один для роботи з даними, а другий для роботи з елементами списку.

Методи для роботи з даними:

- **add(DATA)** – додає дані в кінець списку,
- **add(int, DATA)** – додає дані в зазначене місце списку,
- **addAll(List< DATA>)** – додає діапазон даних в кінець списку,
- **addAll(int, List< DATA>)** – додає діапазон даних в зазначене місце списку,
- **clear()** – очищує список з даних,
- **getItemAt(int)** – дістає дані з заданої позиції в списку.

Всі ці методи викликаються для керування списком. Більш цікавими є методи для роботи з елементами списку, вони є внутрішніми і викликаються самою бібліотекою.

Методи для роботи з елементами списку:

- **onCreateViewHolder(ViewGroup, int)** – викликається бібліотекою для створення елементу списку рівно стільки разів стільки потрібно для того щоб заповнити екран елементами списку. Приймає параметри *ViewGroup* (елемент самого списку в якому розміщуються всі інші елементи) та int (число яке відповідає за тип створеного елементу),
- **onBindViewHolder(VH, int)** – метод який викликається бібліотекою при прокрутці та оновленні даних. Отримує 2 параметри. Першим є віджет елемента, з усіма текстовими полями та необхідними слухачами для дій користувача. Другий це порядковий номер елемента в списку, призначений для того щоб розуміти якими даними потрібно заповнити цей елемент списку. Метод надає елементу його дані для показу через метод у елемента bind() та прив'язує до нього слухачі.

Також в класі присутнє поле data, що являє собою список “сирих” даних для відображення. Саме звідси метод *onBindViewHolder* дістає дані за їх позицією.

Отож цей клас звів роботу зі списками до мінімуму. Все, що вимагається від програміста - це створити (“намалювати”) елемент списку, надати його в адаптер і прив’язати його до *RecyclerView*.

```
private fun initRecyclerView() {
    adapter = BaseRecyclerAdapter { group : ViewGroup , _ : Int  ->
        ArticleVH(group)
    }
    adapter.setOnItemClickListener {
        context?.run {
            SingleArticleActivity.launch( context: this, it)
        }
    }
    articles_recycler_view.layoutManager = LinearLayoutManager(activity)
    articles_recycler_view.adapter = adapter
}
```

Рис. 2.7. Створення та налаштування адаптера

Як можна бачити з рисунку 2.7. Все, що необхідно - це створити об'єкт адаптера та надати йому постачальник елементів. Цей постачальник буде використовуватися методом *onCreateViewHolder* для створення необхідних елементів. Далі встановлюється слухач на натиснення користувачем на елемент. Останніми кроками є налаштування *RecyclerView* та підв'язка до нього адаптера.

Окремої уваги заслуговує абстрактний клас *BaseRecyclerViewHolder*, який є базовим для всіх елементів цього адаптера. Він забирає на себе повторювану логіку усіх елементів та має абстрактний метод *bind(DATA)*, що використовується у методі адаптера *onBindViewHolder*.

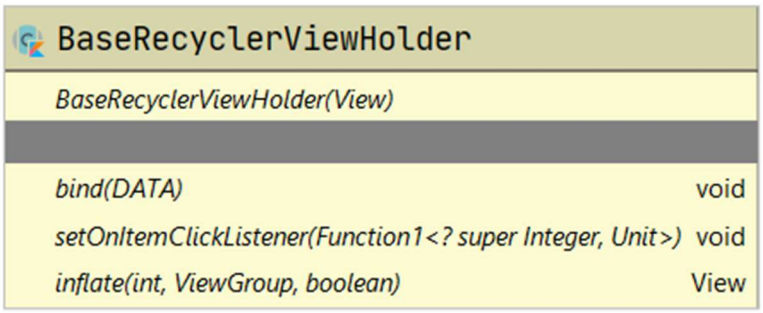


Рис. 2.8. Базовий елемент списків

Отож завдяки базовому класу, написання елементу списку зводиться до простого прив'язування xml файлу до цього класу, та заповнення всіх полів даними в вище зазначеному методі *bind()*

```
class ArticleVH(parent: ViewGroup)
    : BaseRecyclerViewHolder<Article>(
        inflate(
                R.layout.item_article,
                parent,
                attachToRoot: false
        )
) {
    override fun bind(data: Article) {
        itemView.image_article.set(data.imagePath)
        itemView.title_article.text = data.title
        itemView.description_article.text = data.description
    }
}
```

Рис. 2.9. Клас елемента списку

Можна виділити також роботу з завантаженням даних до цього списку під час прокрутки. Цей процес називається пагінацією, і полягає він в тому, що в списку на початку не має всієї вибірки даних. З технічної сторони це реалізовано так, що коли метод *onBindViewHolder* викликається для останнього елемента в завантаженому списку, то спрацьовує слухач, який ініціює завантаження наступної частини даних. По їх завантаженню в фрагменті спрацьовує слухач на їх оновлення й ініціює оновлення адаптера, який в свою сергу показує оновлену вибірку даних.

2.4. Фреймворк Glide

Glide[2] – це фреймворк для роботи з зображеннями який забирає всю монотонну роботу пов'язану з картинками на себе і разом з тим лишає можливість гнучкого налаштування.

```
final Place place = stateData.getSelectedPlace();
Glide.with( fragment: this)
     .load(place.getPhotoUrl())
     .into(placeIcon);
```

Рис. 2.10. Приклад використання фреймворку Glide

Приклади роботи, яку фреймворк *Glide* значно спрощує:

1. завантаження зображення в фоновому потоці,
2. завантаження зображення за вказаним URL адресом,
3. кешування завантаженого зображення,
4. показ прогресу завантаження,
5. стиснення зображення відповідно до розмірів елементу та розширення екрану за для покращення продуктивності роботи додатку,
6. завантаження зображення за вказаним URI в системі Android,
7. накладання фільтрів на зображення,
8. заокруглення кутків зображення,
9. накладання ефектів замилювання,
10. показ файлів формату GIF.

Розглянемо приклад показу прогресу завантаження. У бібліотеці *Glide* для цього потрібно створити “заповнювач”, який буде заповнювати місце картинки до того часу, як вона завантажиться. Для цього в бібліотеці присутні ряд анімацій, одна з яких це “круговий покажчик прогресу”. При його створені можна вказати його радіус, ширину обведення та колір заповнення. Після цього надати створений об'єкт бібліотеці.

```
val drawable = CircularProgressDrawable(context)
        .apply { this: CircularProgressDrawable
            strokeWidth = 5f
            centerRadius = 30f
            setColorSchemeColors(context.getColorSimple(R.color.colorAccent))
            start()
        }
Glide.with(context)
        .load(attachment.file)
        .placeholder(drawable)
        .error(R.drawable.ic_error)
        .into(binding.imgAttachment)
```

Рис. 2.11. Приклад використання заповнювача

Також в бібліотеці дуже просто накладати на зображення фільтри, які також можуть бути скомбіновані між собою з допомогою класу *MultiTransformations*. В даному прикладі розглянемо випадок, коли ми хочемо показати зображення круглим, якщо дані до цього зображення доступні, а якщо не доступні, то показуємо його круглим, в сірих тонах і розмитим. Для таких цілей нам потрібні 3 трансформаційні класи:

- **CropCircleTransformation()**
- **GrayscaleTransformation()**
- **BlurTransformation()**

```
val context = itemView.context
Glide.with(context)
    .load(context.getFile(data.data.mainPicture))
    .apply(
        RequestOptions.bitmapTransform(
            if (data.isEnabled) CropCircleTransformation()
            else MultiTransformation(
                GrayscaleTransformation(),
                CropCircleTransformation(),
                BlurTransformation()
            )
        )
    )
    .into(binding.imgPoint)
```

Рис. 2.12. Приклад використання фільтрів

Якщо користувачу необхідно завантажити якісь конфіденційні дані, які вимагають реєстрації в додатку (наприклад фотографію паспорта), то зазвичай для завантаження таких даних серверна частина вимагає надсилати *JWT* підпис в “голові” запиту. *Glide* також має можливість завантажувати зображення в таких випадках. Якщо ваш запит це звичайний *GET* без авторизації, ви можете просто передати рядок з URL, що веде до зображення. У випадку, якщо потрібна авторизація чи ще якісь додаткові дані можна використати клас *GlideUrl()*, куди необхідно передати саму *URL* та конструктор запиту. В даному випадку розглянемо додавання “шапки” авторизації. Використаємо *LazyHeaders.Builder(),* в який додамо авторизацію у вигляді *JWT* ключа.

```
fun getGlideUrlFor(name: String?): GlideUrl {
    val token = userDataSource.token
    val jwt = token?.jwt
    val bearerToken = token?.bearerJwt ?: "null"
    val userId = token?.decodedJwt?.userId
    val url = "${App.instance?.baseUrl}/api/v1/users/" +
            "$userId/document?filename=$name&jwt=$jwt"
    return GlideUrl(url, LazyHeaders.Builder()
            .addHeader(Constants.AUTHORIZATION_HEADER, bearerToken)
            .build())
}
```

Рис. 2.13. Приклад формування складного запиту

Як раніше було сказано, Glide сам може керувати кешом зображень, які він завантажував. Якщо після завершення сесії необхідно скинути кеш, то розробники передбачили і такий сценарій. Потрібно просто викликати метод *clearDiskCashe()*

```
override fun onDestroy() {
    super.onDestroy()
    context?.let { Glide.get(it).clearDiskCache() }
}
```

Рис. 2.13. Приклад очистки кешу

2.5. Розширення GlideImageView

Попри всі переваги використання бібліотеки *Glide* в ній залишається один суттєвий недолік, це роздільність елемента в який поміщується картинка та самих методів завантаження картинки. Постала необхідність створити клас, який міг би розміститися на екрані у вигляді елементу, і на додачу мав усі переваги *Glide* у завантаженні зображень. Керуючись цими мотивами був створений новий клас *GlideImageView*.

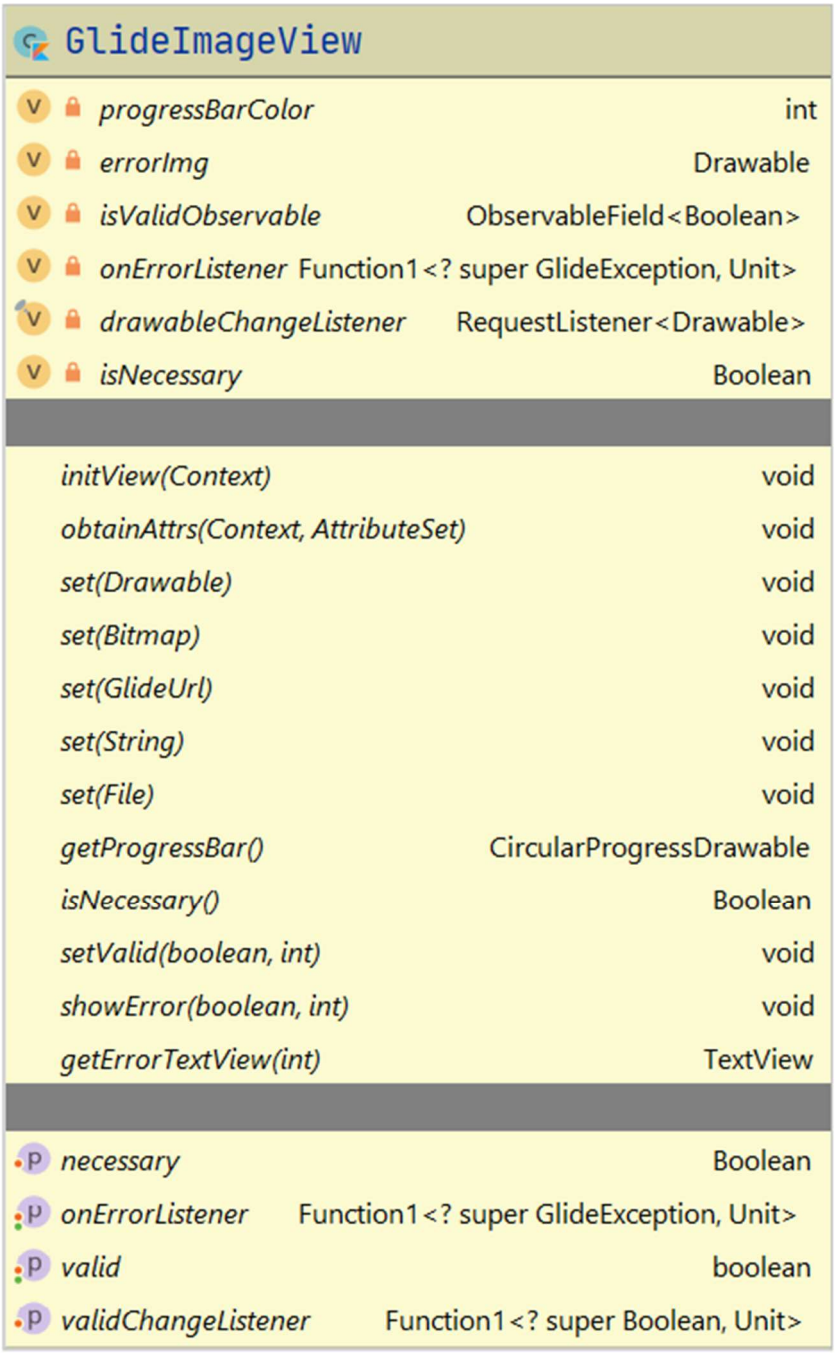


Рис. 2.14. Структура класу *GlideImageView*

Це клас, який наслідує стандартний клас системи Android *ImageView*, має всі його методи, але, в додаток, має декілька перевантажених методів *set(),* які перенаправляють передані дані в бібліотеку Glide, яка вже з допомогою своїх методів розміщує зображення в елементі.

Було написано кілька перевантажень методу *set(),* які приймають такі типи даних:

- **Drawable,**
- **Bitmap,**
- **GlideUrl,**
- **String,**
- **File.**

Розглянемо один з них.

```
fun set(file: File?) {
    Glide.with( view: this)
            .load(file)
            .centerCrop()
            .placeholder(progressBar)
            .error(errorImg)
            .listener(drawableChangeListener)
            .into( view: this)
}
```

Рис. 2.14. Приклад методу *set()*

Даний метод приймає у вигляді параметра об'єкт класу *File*. Розберемо кожен крок цього методу.

1. Ініціюємо роботу бібліотеки методом *with()* і в якості передаємо йому *this* (елемент в якому розміщується зображення), з якого бібліотека пізніше дістане об'єкт класу *Context*.
2. Ініціюємо завантаження файлу викликом метода *load(),* передавши йому об'єкт класу *File*.

3. Викликаємо метод *centerCrop(),* який відцентрує картинку в елементі та обріже зайві її частини, щоб вона зайняла всю площу елемента.
4. З допомогою методу *placeholder()* передаємо бібліотеці об'єкт для показу прогресу завантаження зображення.
5. Методом *error()* передаємо картинку, яка буде показана у випадку помилки завантаження зображення з файлу.
6. Реєструємо слухача стану завантаження зображення для додаткової логіки даного класу, з допомогою методу *listener().*
7. Методом *into()* вказуємо куди завантажувати зображення, і відповідно передаємо параметр *this*, щоб завантаження відбулося в елемент, на якому був викликаний метод *set().*

2.6. Фреймворк Retrofit

Retrofit [3] - це REST клієнт для мови програмування Java та операційної системи Android. Це фреймворк, який значно спрощує роботу з завантаженням або вивантаженням даних через *REST* сервіси. В Retrofit ви можете налаштувати серіалізацію та десеріалізацію для кожного окремого типу даних. Зазвичай для того, щоб передавати дані у вигляді *JSON* використовують бібліотеку *Gson*, але програміст може сам налаштувати процес конвертації, навіть для тих даних, які приходять в *XML* форматі. Для виконання HTTP запитів цей фреймворк використовує бібліотеку *OkHttp*.

Для налаштування роботи Retrofit проходить у 3 кроки.

1. Створити моделі. Це класи – репрезентації даних, які надсилаються на HTTP сервіс чи отримуються з нього. За своїм змістом це класи, які є посередниками між мовою програмування додатку та “мовою”, якою спілкується додаток з сервісом.
2. Створення інтерфейсу з переліком усіх HTTP операцій. Пізніше Retrofit згенерує імплементацію для цього інтерфейсу, яка робитиме самі запити на сервіс, а для програміста залишиться

тільки викликати методи цього інтерфейсу і отримувати у відповідь підготовані дані з відповіді.

3. Налаштування Retrofit через клас *Retrofit.Builer*. Йому потрібно надати об'єкт класу *OkHttpClient* та фабрику з конвертації об'єктів *Gson*.

Розглянемо створення та налаштування *OkHttpClient*.

```
final OkHttpClient client =
        new OkHttpClient()
                .newBuilder()
                .connectTimeout( timeout: 40, SECONDS)
                .readTimeout( timeout: 60, SECONDS)
                .writeTimeout( timeout: 60, SECONDS)
                .build();
```

Рис. 2.15. Налаштування *OkHttpClient*

Для його створення ніяких особливих параметрів не потрібно. Розберемо кожен крок створення.

- Перші 3 рядки ініціація самого будівельника класу, який тимчасово триматиме дані поки ті налаштовуються, адже як тільки *OkHttpClient* створено, багато параметрів змінити вже не можна.
- Встановлюється максимальний час на з'єднання, метод приймає першим параметром цифру і другим значення, що символізує одиниці виміру попередньої цифри.
- Встановлюється максимальний час на читання, метод приймає аналогічні параметри до попереднього.
- Встановлюється максимальний час для запис, метод приймає аналогічні параметри до попереднього.
- Ініціюється *OkHttpClient* на основі параметрів, які були передані будівельнику. Далі об'єкт цього класу буде використовуватися згенерованим класом Retrofit в якості постачальника HTPP з'єднання.

Будівельник має ще декілька корисних методів, один з яких я б хотів розглянути.

```
private final OkHttpClient client = new OkHttpClient.Builder()
        .connectTimeout( timeout: 1, MINUTES)
        .writeTimeout( timeout: 1, MINUTES)
        .readTimeout( timeout: 1, MINUTES)
        .addInterceptor(this::interceptor)
        .addInterceptor(logging)
        .build();
```

Рис. 2.16. Налаштування *OkHttpClient*, метод *addInterceptor()*

Цей метод використовується для того, щоб “вклинитися” в момент, коли дані готові до відправки, але ще не відправлені, або момент коли дані вже прийшли, але ще нікуди не передавалися.

Цей метод можна використати для перевірки на відповіді сервера з кодом 401, який означає, що ключ, яким підписуються запити, застарів і його необхідно оновити і повторити запит.

Також з допомогою цього методу можна організувати логування запитів.

Розглянемо створення *Gson* фабрики.

```
public static GsonBuilder getGsonBuilder() {
    final GsonBuilder gsonBuilder = new GsonBuilder().setLenient();
    Timestamp.registerTypeAdapter(gsonBuilder);
    Pdf.registerTypeAdapter(gsonBuilder);
    TokenTariff.registerTypeAdapter(gsonBuilder);
    return gsonBuilder;
}
```

Рис. 2.17. Приклад створення *Gson* фабрики

Створення проходить в декілька етапів.

- Ініціалізація будівельника.
- З допомогою методу *setLateInit()* кажемо фабриці ініціалізуватися лише коли вона стане потрібною, це зроблено для того, щоб розвантажити процес ініціалізації *Retrofit*.
- Реєструємо декілька типів даних які мають особливу конвертацію.

```
public static void registerTypeAdapter(GsonBuilder builder) {
    builder.registerTypeAdapter(Timestamp.class,
            (JsonSerializer<Timestamp>) (src, typeOfSrc, context1) ->
                    src == null
                            ? null
                            : new JsonPrimitive(src.getTimestamp()))
            .registerTypeAdapter(Timestamp.class,
                    (JsonDeserializer<Timestamp>) (json, typeOfT, context) -> {
                        final String inString = json.getAsString();
                        return inString == null || "0001-01-01T00:00:00Z".equals(inString)
                                ? null
                                : new Timestamp(inString);
                    });
}
```

Рис. 2.18. Приклад реєстрації класу *Timestamp* до *GsonBuilder*

Коли при десеріалізації *Gson* фабрика бачить клас, який в ній зареєстрований як особливий. Вона використовує той код, що їй було передано при реєстрації цього типу даних.

При реєстрації передається дві лямбди. Перша для серіалізації, друга, відповідно, для десеріалізації.

Отож *JSON* об'єкт повинен бути записаний у вигляді рядка. При серіалізації об'єкта ми повертаємо його рядкове представлення, а при десеріалізації ми переводимо його рядкове представлення в об'єктне.

Тепер, коли всі моменти, що передують створенню *Retrofit.Builer*, з'ясовано, можна перейти до самого будівельника *Retrofit*.

```
return new Retrofit.Builder()
        .baseUrl(Constants.DIPLOM_API_BASE_URL)
        .client(client)
        .addConverterFactory(GsonConverterFactory.create(gson))
        .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
        .build();
```

Рис. 2.18. Приклад реєстрації класу *Timestamp* до *GsonBuilder*

- Ініціюємо *Retrofit.Builer*.
- Встановлюємо базову URL адресу.
- Додаємо *Gson* фабрику.

- Додаємо фабрику для роботи з бібліотекою *RxJava*, про яку буде згадано трохи згодом.

Розглянемо створення інтерфейсу з переліком усіх HTTP операцій.

```
public interface DiplomApi {

    @POST("directions")
    Observable<List<RouteResponse>> getDirections(
            @Header("Content-Type") String contentType,
            @Body DirectionRequest directionRequest);

    @GET("/api/directions/lookup-place-id/{latitude}/{longitude}")
    Observable<PlaceDescription> getPlaceDescription(
            @Path("latitude") double latitude,
            @Path("longitude") double longitude);
}
```

Рис. 2.19. Приклад інтерфейсу з HTTP операціями

Вибір HTTP метода обирається з допомогою анотацій:

- **@GET,**
- **@POST,**
- **@PUT,**
- **@DELETE,**
- та інші.

Їм в параметр передається кінцева частина URL (URL без базової частини), та за допомогою символів “{” та “}”, позначаються місця, які мають бути замінені якимись даними.

Також є функціональні анотації, які впливають на подальшу роботу цього методу, як от ,наприклад, анотація *@Streaming*. Зазвичай *Retrofit* спершу повністю отримує відповідь з серверу, зберігаючи її при цьому в операційну пам'ять пристрою, а потім передає її як відповідь метода. Ця анотація була зроблена для того, щоб *Retrofit* одразу передавав дані у відповідь як потік даних, на випадок, якщо цей метод призначений для завантаження великих файлів, і програміст хоче, щоб збереження йшло безпосередньо до файлу.

Параметри метода також позначаються анотаціями.

- **@Header(“value”)** – цей параметр надалі буде записано в шапку запиту, як значення запису, а його ключ буде взятий з параметра анотації.
- **@Body** – позначає параметр як той, що буде на пряму записаний в тіло запиту,
- **@Path(“value”) –** позначає параметр як той, що буде записаний до URL, замість відповідного йому “{value}”
- **@Query(“value”) –** позначає параметр як той, що буде записаний до URL у вигляді query, у якості ключа -параметр анотацій, у якості значення- параметр, який маркує анотація.
- **@Url –** у випадку, якщо URL для цього метода потрібно змінюватися, її можна передати параметром метода, який маркується цією анотацією.

Після завершення запиту його результат буде десеріалізований у той об'єкт, який прописаний в методі інтерфейса, як результат. У випадку, якщо результуючий об'єкт в інтерфейсі, загорнутий в один із об'єктів *RxJava* (*Observable, Single, Maybe, Call*), буде використана фабрика, яку ми додали при створенні Retrofit.

Наступним етапом хотілося б розглянути моделі. Отож, як було сказано раніше, моделі це об'єкти, які є посередниками між мовою програмування додатку та “мовою” узгодженою з сервісом. Сервісом може бути як web-сервер, так і локальна база даних. Моделі часто використовуються як класи для групування даних, які мають одне призначення в бізнес логіці проекту, чи навіть при передачі даних всередині додатку, коли вони мають спільну току призначення.

Розглянемо модель *Place*.

```
public final class Place
        implements Parcelable {

    @SerializedName(value = "lat",
            alternate = {"latitude", "Latitude"})
    private Double latitude;
    @SerializedName(value = "lng",
            alternate = {"longitude", "Longitude"})
    private Double longitude;
    @SerializedName("id")
    private String id;
    @SerializedName("name")
    private String name;
    @SerializedName("address")
    private String address;
```

Рис. 2.20. Приклад моделі

В даному випадку вона використовується як для десеріалізації відповіді з сервера, так і для збереження місць у локальну базу даних смартфону.

Ми можемо бачити, що в ній є декілька полів з анотаціями.

Анотація *@SerializedName("value")* призначена для маркування поля над яким вона написана. Вона тут знаходиться для бібліотеки *Gson*, щоб коли бібліотека читає або створює JSON файл, вона розуміла, який ключ у *JSON* файлі відповідає якому значенню в моделі. По суті весь процес переходу між "мовами" зав'язаний на цій анотації.

Анотації має два параметри: стандартний *value*, який використовується при серіалізації та десеріалізації, та додатковий *alternate*, який приймає список рядків. Він використовується тільки при десеріалізації. Параметр створений для того, щоб зробити десеріалізацію більш гнучкою. Коли *Gson*, під час десеріалізації не знаходить в *JSON* поле з ключем прописаним у *value*, він починає шукати в цій частині *JSON* ключі, які співпадають з альтернативними назвами.

2.7. Фреймворк RxJava

RxJava [4] - це дуже потужний інструмент для реактивного програмування. Це тип програмування, коли слухач реагує на зміну даних, як ті з'являються. Reactivex – проект, який описує себе як комбінація найкращого з патерну Observer, патерну Iterator та функціонального програмування. RxJava – це реалізація цього концепту на мові програмування Java. Цей фреймворк надає можливість програмувати асинхронно, з можливістю підписуватись на оновлення потоку даних.

Блок коду RxJava зазвичай складається з трьох основних частин.

- *Observable* – частина коду, яка означає собою джерело даних, як тільки вони оновлюються запускається увесь наступний ланцюг коду.
- Множина методів для модифікування та редагування даних отриманих з джерела.
- Підписка. Ця частина запускає в роботу оновлення даних у джерелі та підписується на них, після проходження усіх методів модифікації дані потрапляють у цей блок.

```
public void getNearestPlace(Location location,
                            Consumer<Place> consumer,
                            Consumer<Boolean> loadingConsumer) {
    addDisposable(placeDataSource
            .getPlaceDescription(
                    location.getLat(),
                    location.getLng()) Observable< PlaceDescription>
            .map(description -> mapDescription(
                    location,
                    description)) Observable< Place>
            .compose(new AsyncTransformer<>())
            .doOnSubscribe(d -> loadingConsumer.accept( t: true))
            .doAfterTerminate(() -> loadingConsumer.accept( t: false))
            .subscribe(consumer, new RxErrorAction(view)));
}
```

Рис. 2.21. Приклад використання RxJava

Розберемо детально наведений приклад.

- У метод *addDisposable()* передаємо результат підписки, для чого це зроблено буде розписано трохи згодом.
- З *placeDataSource* за допомогою методу *getPaceDesctiption()* дістаємо *Observable* із загорнутим у нього об'єктом *PlaceDescription*. Цей метод усередині викликає звернення до API з допомогою раніше описаного інтерфейсу Retrofit. І як тільки Retrofit виконає запит, він помістить в оговорений *Observable* відповідь, що спричинить подальше виконання ланцюга коду.
- Метод *map* створений для того, щоб перетворювати дані з одного вигляду в інший, у даному випадку він викликає метод, щоб перетворити модель *PlaceDescription* на *Place*.
- Метод **compose** викликається, щоб трансформувати потік даних. У даному випадку він робить так, щоб звернення до API та перетворення об'єкта відбувалось у фоновому потоці, а коли надійде готовий об'єкт *Place*, ланцюг коду продовжить своє виконання в UI потоці. Про цей метод та переходи між потоками, також буде написано трохи згодом.
- Метод *doOnSubscribe()* викликається відразу після початку виконання ланцюга коду. У даному випадку він оновлює слухача, який відповідає за показ прогресу завантаження, і ставить йому значення, яке відповідає за показ прогресу.
- Метод *doAfterTerminate()* викликається відразу після виконання коду, що написаний в методі *subscribe()*. У даному випадку він оновлює слухача, який відповідає за показ прогресу завантаження. І ставить йому значення, яке відповідає за закінченню завантаження.

- Метод *subscribe()* – це той метод, який запускає в роботу увесь ланцюг коду. Він має доволі широкий функціонал, тому доречно буде розписати про нього окремо.

Розберемо, навіщо на прикладі використано метод *addDisposable()*. Справа в тім, що підписка відбувається в *Presenter*, а в методі *subscribe()* ми звертаємося до *View*. Може бути такий випадок, коли *View* "попросило" *Presenter*-а надати якісь дані, а потім відв'язалося від нього. Це спричинить те, що коли ланцюг коду дійде до метода *subscribe(),* поле view вже не буде вказувати ні на який об'єкт, що спричинить помилку виконання коду і "падіння" додатку. Таке може статися, наприклад, коли користувач натиснув на кнопку оновити і згорнув додаток, чи заблокував телефон. Ці дії спричиняють відписку *Presenter*-а від *View*.

Отож метод *addDisposable()* додає результат підписки (*Disposable*) до спеціального списку, який при відв'язці *View* від *Presenter* очищається та перериває всі записані в нього *Disposable*.

Disposable – це не дані, це посилання на саму підписку. У цього класу є лише два методи:

- **isDisposed()** – показує чи дана підписка ще виконується,
- **dispose()** – перериває виконання коду в ланцюгу на який була здійснена ця підписка. Метод чимось схожий до методу **Thread.interrupt()**.

Отже коли список з *Disposable* очищено, та кожну підписку перервано, додаток не "впаде" з тої причини, що коли підписка "захоче" звернутися до **View**, його вже не буде.

Методу *compoce()* в наведеному прикладі передається клас *AsyncTransformer*. Це метод, який може застосувати до Observable одразу кілька трансформацій та допоможе дотримуватися методології DRY.

Розглянемо клас *AsyncTransformer*. Він наслідує інтерфейс *ObservableTransformer* і визначає один його метод *apply(),* який викликаєтеся

методом *compose()*. Вся трансформація відбувається в цьому методі. Ми отримуємо *Observable* і трансформуємо його.

```
public class AsyncTransformer<T>
        implements ObservableTransformer<T, T> {

    @Override
    public ObservableSource<T> apply(Observable<T> upstream) {
        return upstream
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread());
    }
}
```

Рис. 2.22. Клас *AsyncTransformer*

Тут викликаються два методи.

- **subscribeOn()** – метод, який приймає *Scheduler* та застосовує його до самого створення *Observable*. Відповідно, якщо після цього *Scheduler* не змінювати, то метод *subscribe()* також лишиться в визначеному потоці.
- **observeOn()** – метод, який також приймає *Scheduler*, але застосовує його тільки тоді, коли ланцюжок виконання коду дійде до цього методу. Тобто це метод, який може переміщувати виконання ланцюжка між потоками.

Scheduler-ів існує декілька [5].

- **IO** – призначений для роботи з запису та читання даних. Його використовують найчастіше. Являє собою необмежений пул потоків.
- **SINGLE** – має тільки один потік, немає значення скільки разів на нього підписатися, всі *Observable* будуть виконуватися лише в одному потоці.
- **COMPUTATION** - призначений для роботи з запису та читання даних. Являє собою пул потоків обмежений кількістю логічних ядер.

- **TRAMPOLINE** – виконує код в поточному потоці шляхом блокування інших операцій.
- **NEW_THREAD** – як можна зрозуміти з назви, створює новий потік кожного разу як його викликають.
- **MAIN_THREAD** – як можна зрозуміти з назви, виконує роботу в головному потоці системи Android (UI)

Розглянемо метод *subscribe()*. Метод повертає об'єкт *Disposable* (було розглянуто раніше) та має декілька перевантажень, у найширшій свої версії має чотири лямбда параметри:

- **onNext** – код буде викликатися щоразу, коли надходять дані з **Observable**,
- **onError** – код буде викликатися щоразу, коли в ході виконання ланцюжка коду виникає помилка,
- **onComplete** – код буде викликано після успішного закінчення виконання ланцюжка,
- **onSubscribe** – код буде викликано після старту виконання ланцюжка.

У RxJava, окрім *Observable*, є ще декілька класів-обгорток. Розглянемо в чому між ними різниця.

- **Single** – це *Observable*, який може надати лише один результат або видати помилку за його відсутності.
- **Maybe** – це *Observable*, який може надати лише один результат і допускає те, що результату може взагалі не бути.
- **Completable** – це *Observable*, який не надає результатів взагалі, тільки сам факт успішного чи не успішного виконання.
- **Observable** – надає один або декілька результатів, найбільш універсальний серед усіх, має метод *onNext* для відлову кожного результату.

- **Call** – розроблений спеціально для запитів на сервер. Може отримати з сервера відповідь та тіло відповіді в різні моменти часу.

Розглянемо ще декілька методів для класу *Observable***.**

- **map** – метод для переробки даних з одного типу в інший.

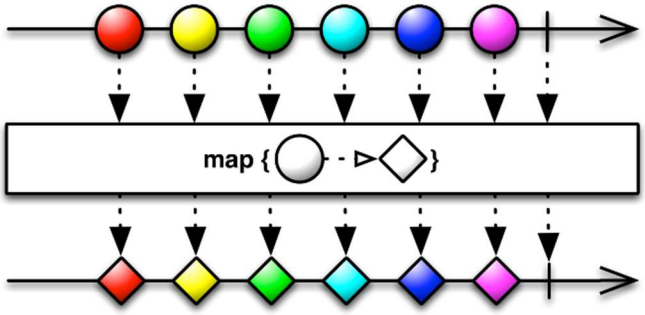


Рис. 2.23. Метод *map* [9]

- **flatMap** – на основі даних з *Observable***,** на якому був викликаний, повертає новий *Observable*

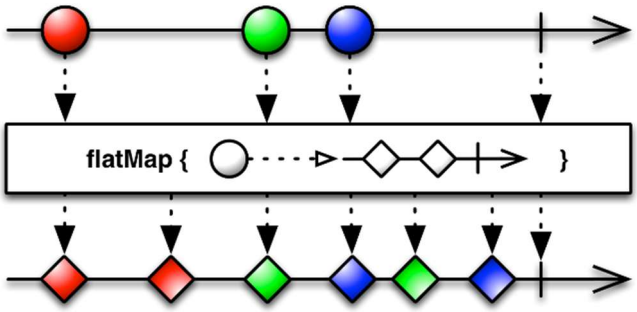


Рис. 2.24. Метод *flatMap* [9]

- **zip** – комбінує результати декількох *Observable* в один.

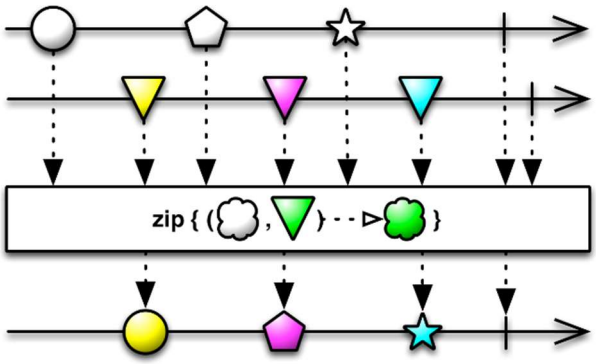


Рис. 2.25. Метод *zip* [9]

- **any** – перевіряє, чи хоча б один з результатів задовольняє предикату.

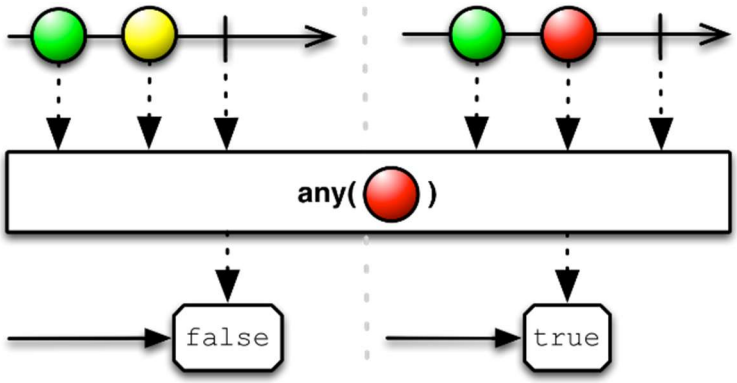


Рис. 2.26. Метод *any* [9]

- **all** – перевіряє, чи всі результати задовольняють предикату

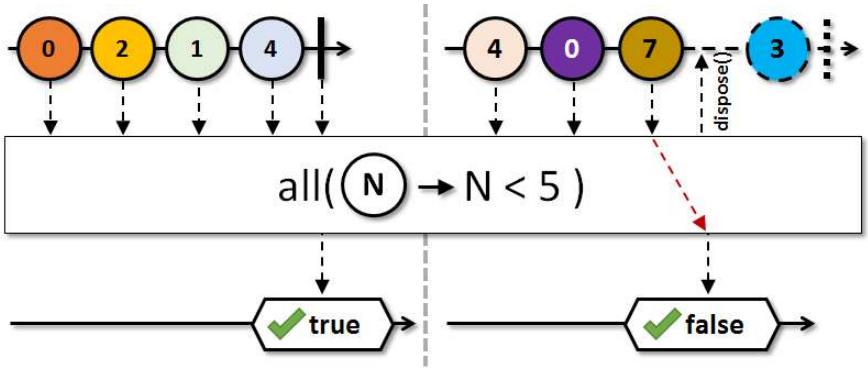


Рис. 2.27. Метод *all* [9]

- **filter** – пропускає результат далі тільки у випадку, якщо він задовольняє предикату.

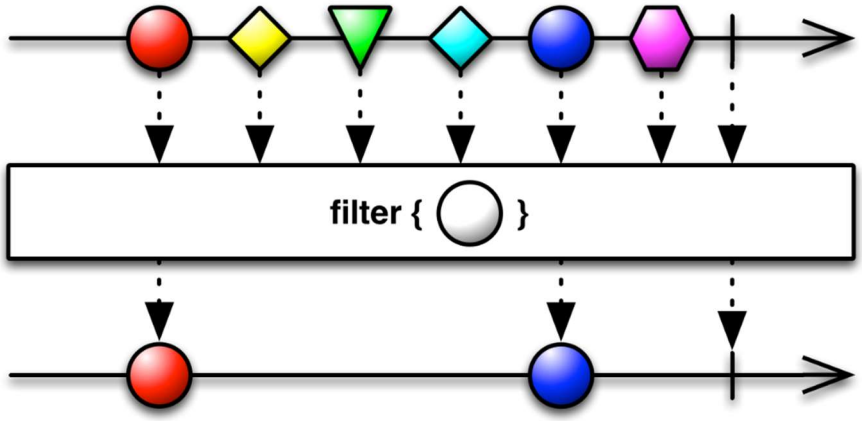


Рис. 2.28. Метод *filter* [9]

- **distinct** – відфільтровує всі результати, які вже надходили.

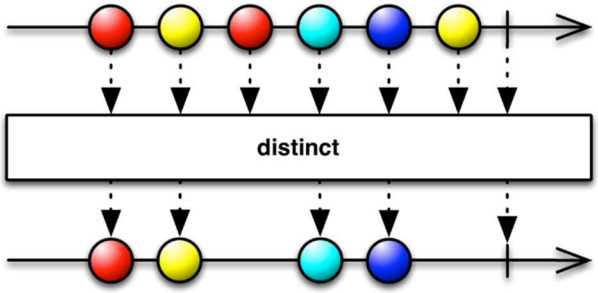


Рис. 2.29. Метод *distinct* [9]

- **debounce** – відфільтровує зміни результату, які стаються надто часто (наприклад, коли користувач вводить дані в пошуковий рядок, ми не хочемо робити пошук з кожною новою літерою, ми хочемо почекати доти доки користувач призупиниться).

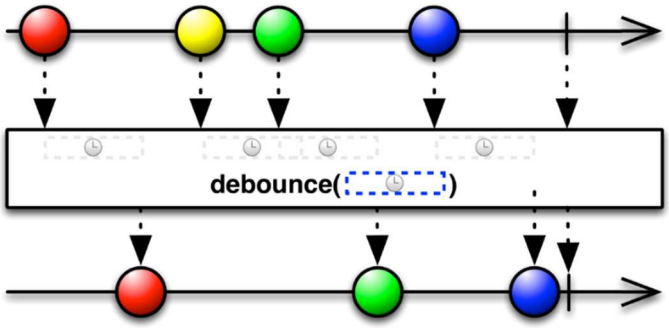


Рис. 2.30. Метод *debounce* [9]

- **cast** – як можна здогадатися з назви, виконує каст результату до заданого в параметрі класу.

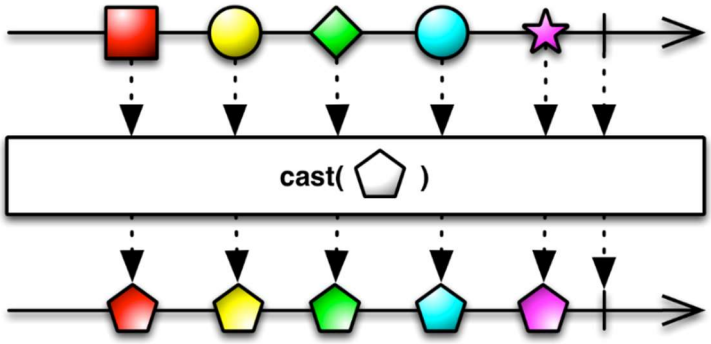


Рис. 2.31. Метод *cast* [9]

- **count –** повертає кількість результатів, яку надав *Observable*.

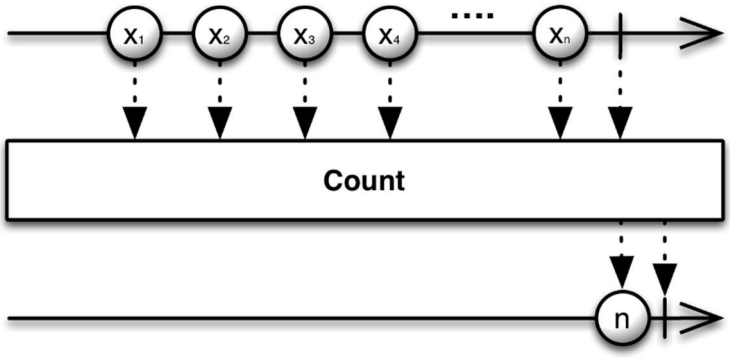


Рис. 2.32. Метод *count* [9]

- **blockingFirst –** блокуючи поточний потік, чекає на надходження результату, повертаючи його.

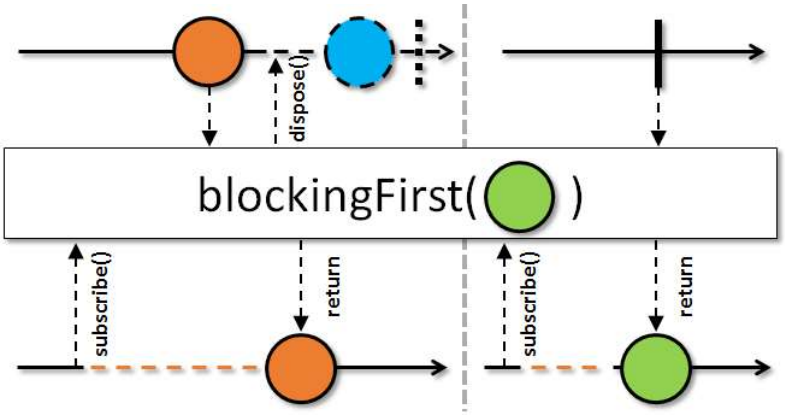


Рис. 2.33. Метод *blockingFirst* [9]

- **forEach –** виконує задану функцію для кожного результату.

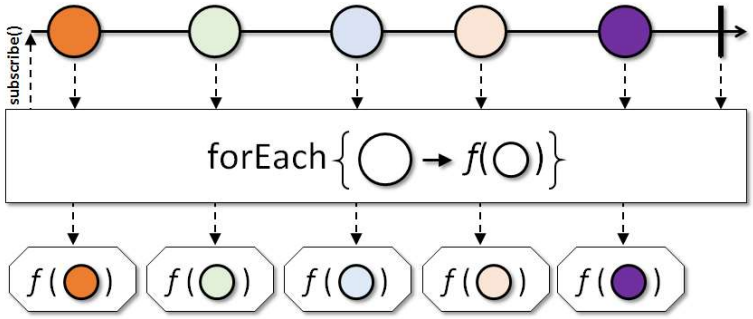


Рис. 2.34. Метод *forEach* [9]

ВИСНОВКИ ДО РОЗДІЛУ 2

В даному розділі було розглянуто засоби та фреймворки використані при розробці даної роботи. Був розглянутий архітектурний патерн, яким я користувався при розробці додатку, та бібліотеки, якими я користувався. Також були розглянуті написані мною модулі, які полегшували розробку даного додатку. Для найцікавіших (на мою думку) із засобів програмування було розглянуто приклади їх використання та трішки внутрішньої структури. Було описано деталі користування деякими бібліотеками та що саме потрібно для їх коректної роботи.

РОЗДІЛ 3

ІНСТРУКЦІЯ З ВИКОРИСТАННЯ ДОДАТКУ

3.1. Головний екран

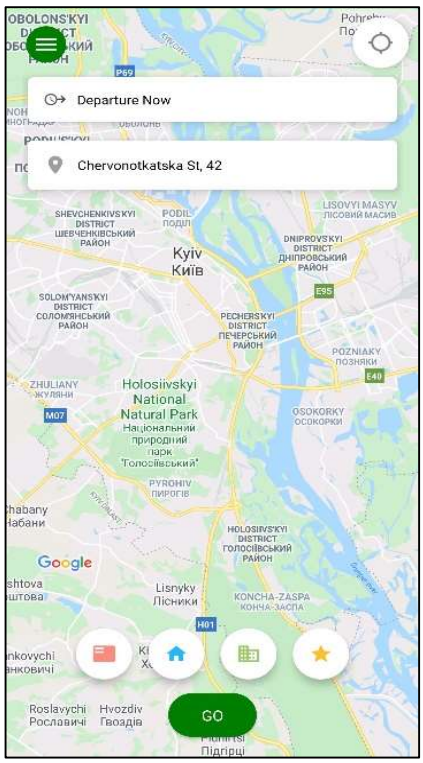


Рис. 3.1 Головний екран

Як тільки користувач завантажує додаток, перше, що він побачить, - це головний екран.

Згори можна бачити 2 кнопки: меню та знайти мене.

Трішки нижче 2 поля для введення. Після натиснення на перше, користувач перейде на екран вибору часу. Після натиснення на друге, користувач потрапить на екран вибору адрес початку та кінця маршруту.

В низу екрану можна бачити 5 кнопок. Кнопка “Go” ініціює побудову маршруту. Кнопка з виглядом картки ініціює побудову маршруту до найближчої записаної на сьогодні події. Кнопка з будиночком переводить на екран вибору адрес з обраною адресою вашого дому в якості фінішу. Кнопка з офісною будівлею робить ре ж саме, що і кнопка з будиночком тільки ,замість адреси дому, обирає адресу роботи. Кнопка з зірочкою робить аналогічну дію до двох попередніх, тільки з адресою найпопулярнішого місця користувача.

3.2. Екран вибору адреси

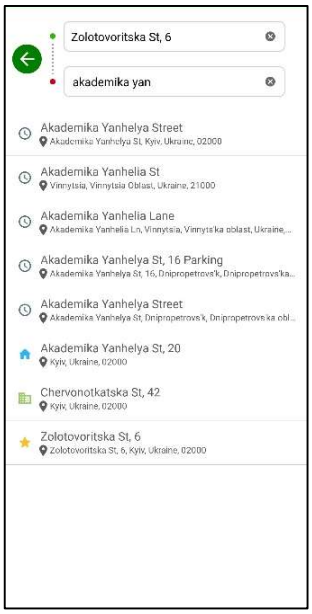


Рис. 3.2 Екран вибору адреси

На даному екрані відображаються місця роботи, дому, 5 найчастіше запитуваних місць, та результати пошуку. Як тільки місця обрано додаток автоматично закриє цей екран.

3.3. Екран вибору часу

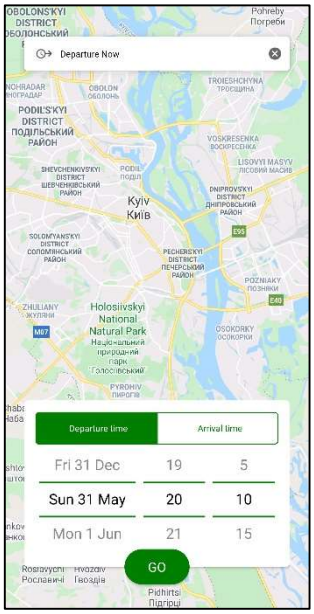


Рис. 3.3 Екран вибору часу

На даному екрані можна обрати час та визначити, чи це час прибуття, чи відправлення

3.4. Меню

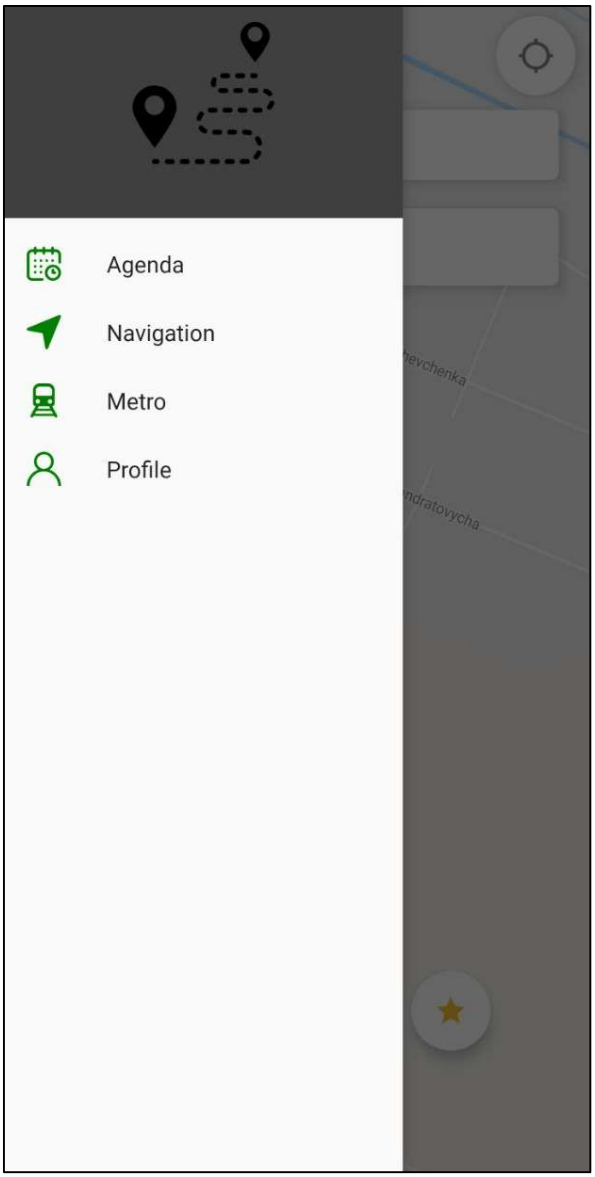


Рис. 3.4 Екран меню

Звідси можна потрапити до календаря, навігації, екрану метро та профіля. В цей екран можна потрапити також лише з перерахованих екранів.

3.5. Екран вибору маршруту

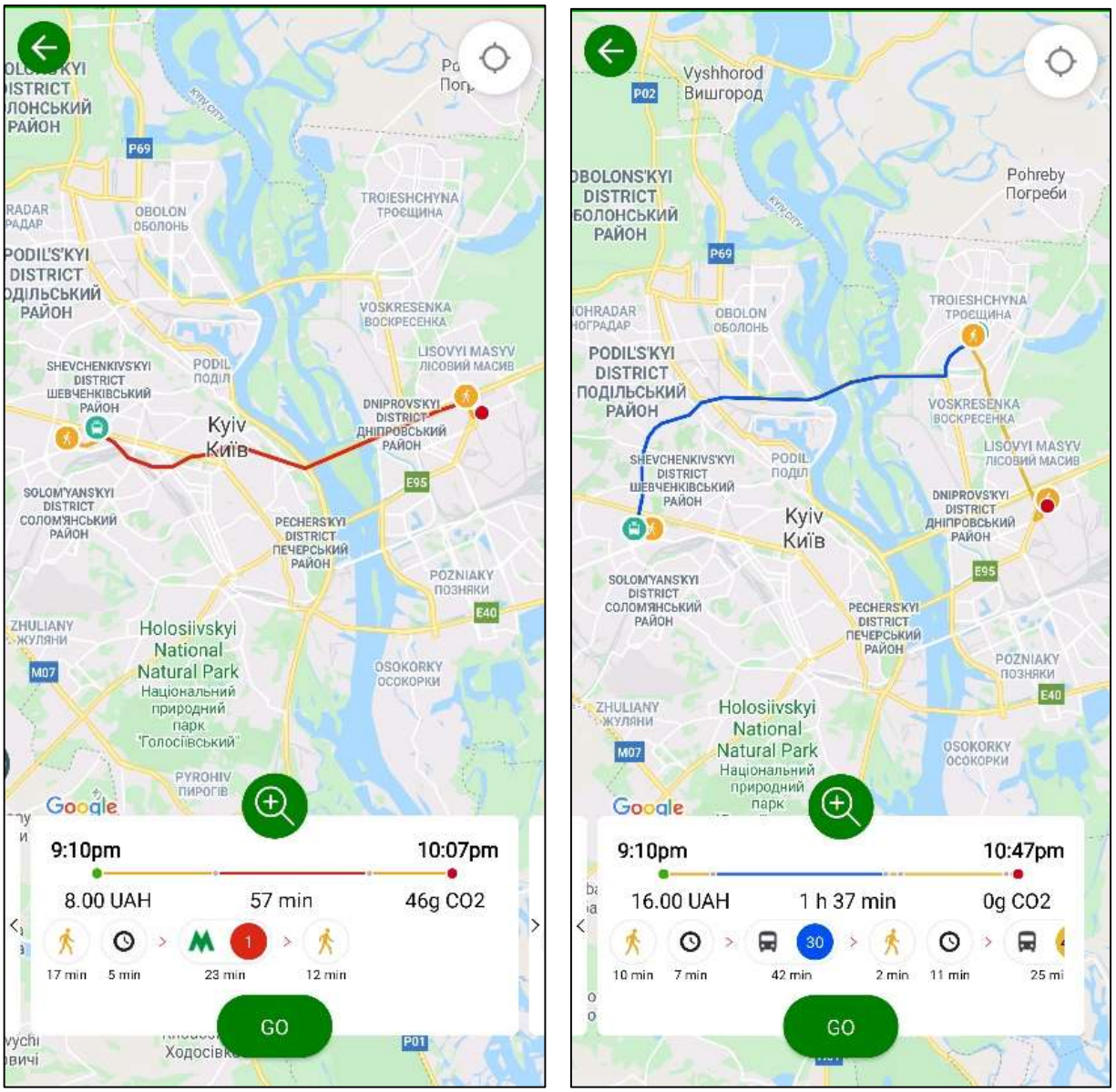


Рис. 3.5 та 3.6 Вибір маршруту

На даних екранах можна бачити короткі відомості про маршрут, кнопку для переходу на екран деталей маршруту (лупа) та кнопку “Go” для початку навігації по обраному маршруту.

3.6. Екран деталей маршруту

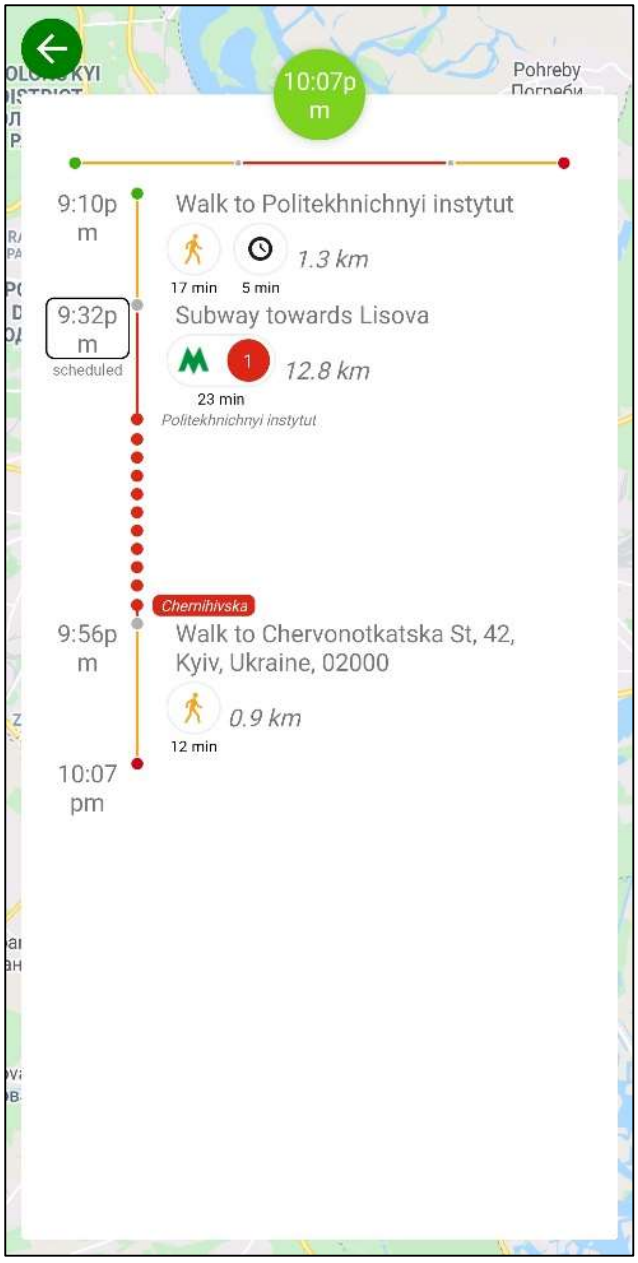


Рис. 3.7 Деталі маршруту

Після натиснення на іконку лупи на екрані вибору маршруту ми потрапимо на екран з деталями маршруту.

Тут відображається інформація, коли користувач прибуде, і в який час в нього будуть пересадки. Також тут описано, якими транспортними засобами потрібно пересуватися і на яку відстань.

3.7. Екран редагування маршруту

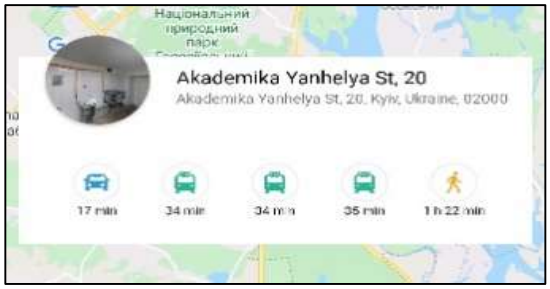


Рис. 3.8 Вибір типу транспорту

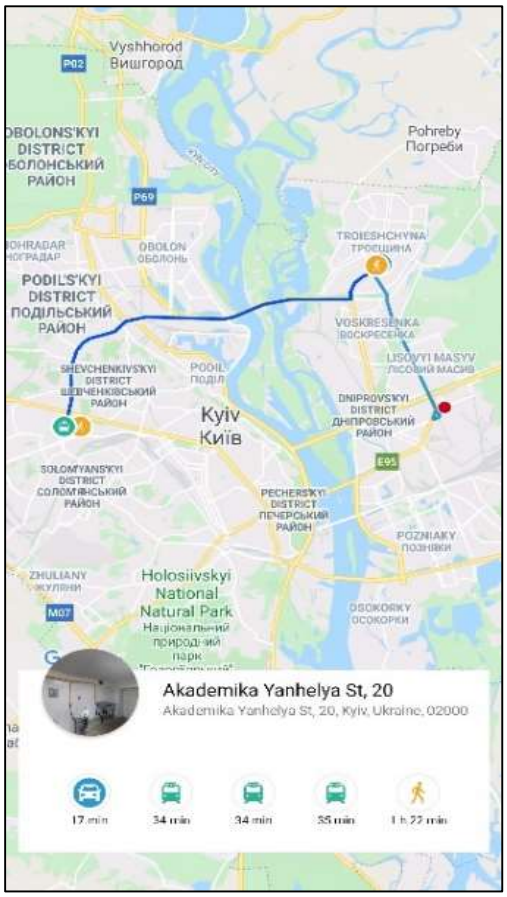


Рис. 3.9 Тип транспорту

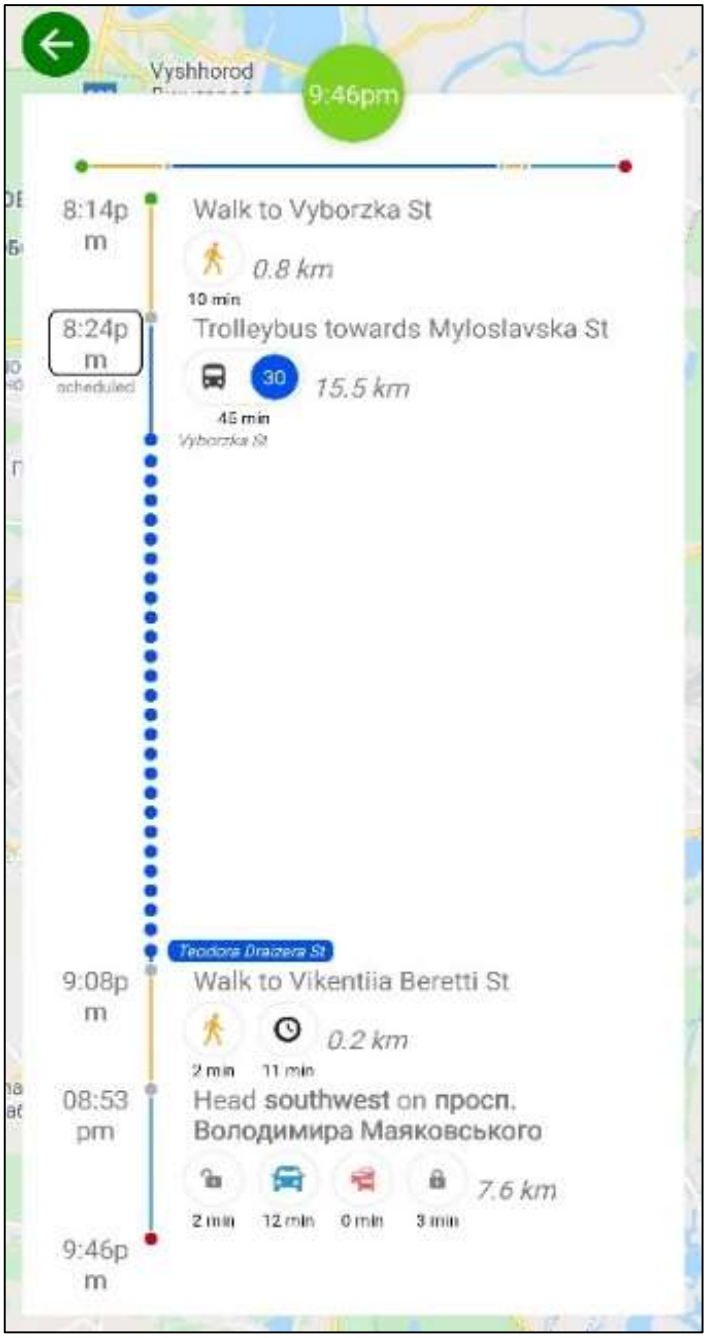


Рис. 3.10 Деталі нового маршруту

Якщо натиснути на маркер на карті, можна потрапити до редагування цього сегменту дороги. На вибір буде надано декілька варіантів, серед яких можна обирати натискаючи на них. Щоб підтвердити зміну, потрібно ще раз натиснути на обраний вид транспорту.

Коли транспорт обрано, можна перейти на деталі маршруту та оцінити новий маршрут. (Рис. 3.10)

3.8. Екран навігації

Коли з маршрутом визначилися, то можна починати навігацію по ньому. Якщо маршрут складається з одного кроку, наприклад їзда на автомобілі, то додаток одразу перейде на екран навігації (Рис. 3.11). Якщо маршрут включає декілька кроків, наприклад ходьбу, пересадки між громадським транспортом, то додаток покаже розширену версію детального екрану з кнопками навігації по кожному кроку (Рис. 3.12).

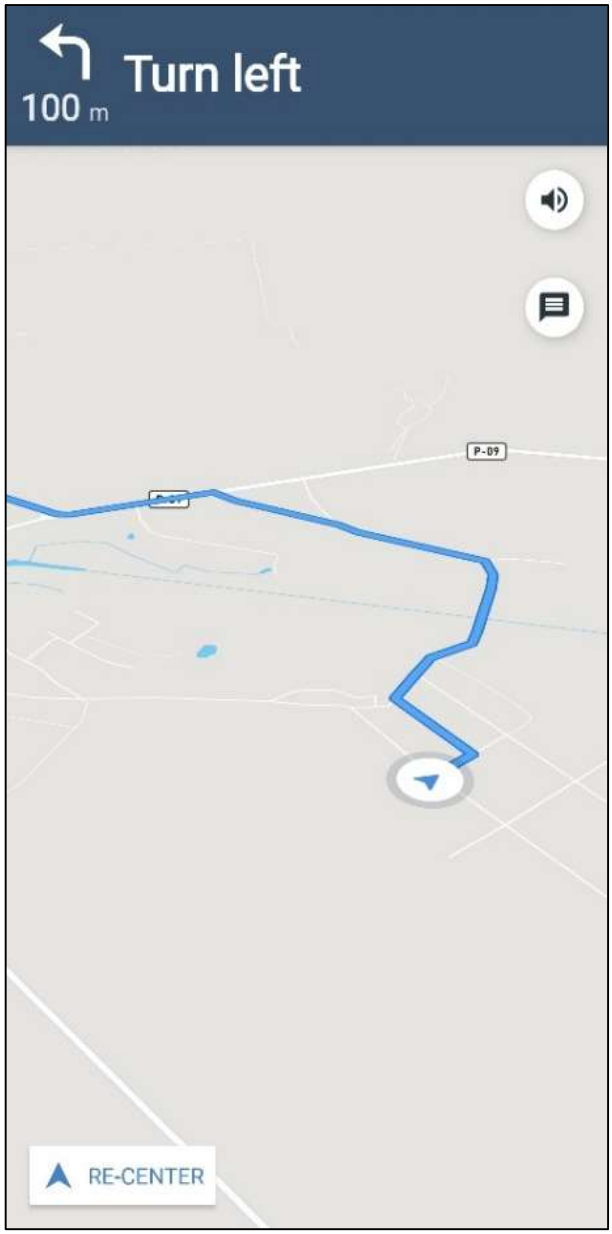


Рис. 3.11 Навігація

Рис. 3.12 Розширена версія детального екрану

3.9. Екран профіля

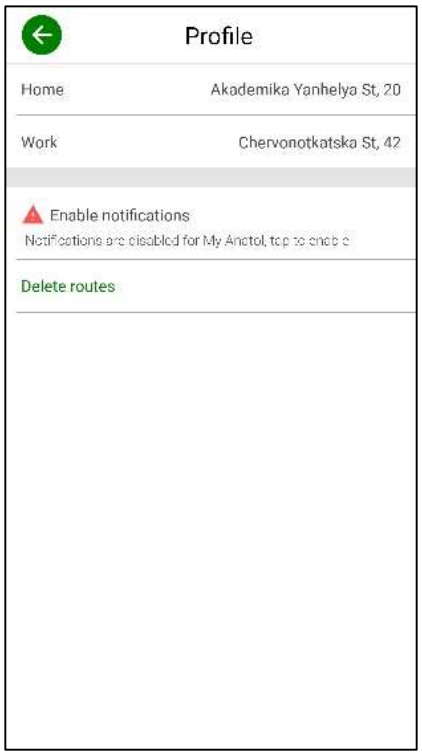


Рис. 3.13 Профіль

В профілі можна вручну обрати місце дому та роботи (Рисунок 3.14), якщо вам не подобається те, яке додаток підібрав автоматично. Вибране мами місце буде пріоритетним перед тим, що було вибрано автоматично.

Також тут можна керувати доступністю сповіщень з календаря та видаляти збережені місця в локальній базі даних.

3.10. Екран вибору місця

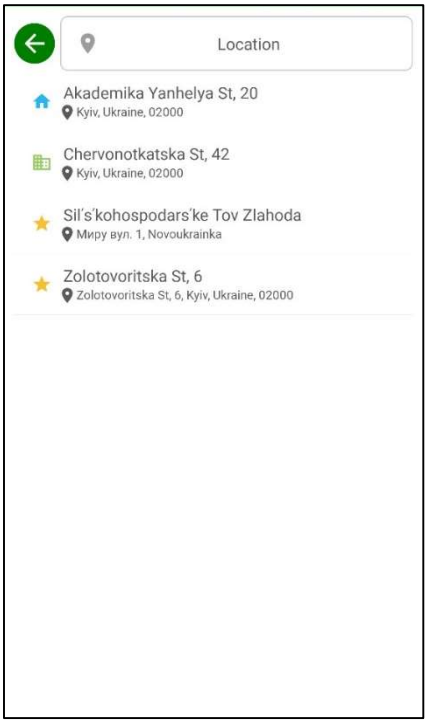


Рис. 3.14 Вибір місця

При введенні тексту в текстове поле, запускається автоматичний пошук місць з подібною назвою. Також ви можете зазначити місце з вже використовуваних вами.

3.11. Точка на карті

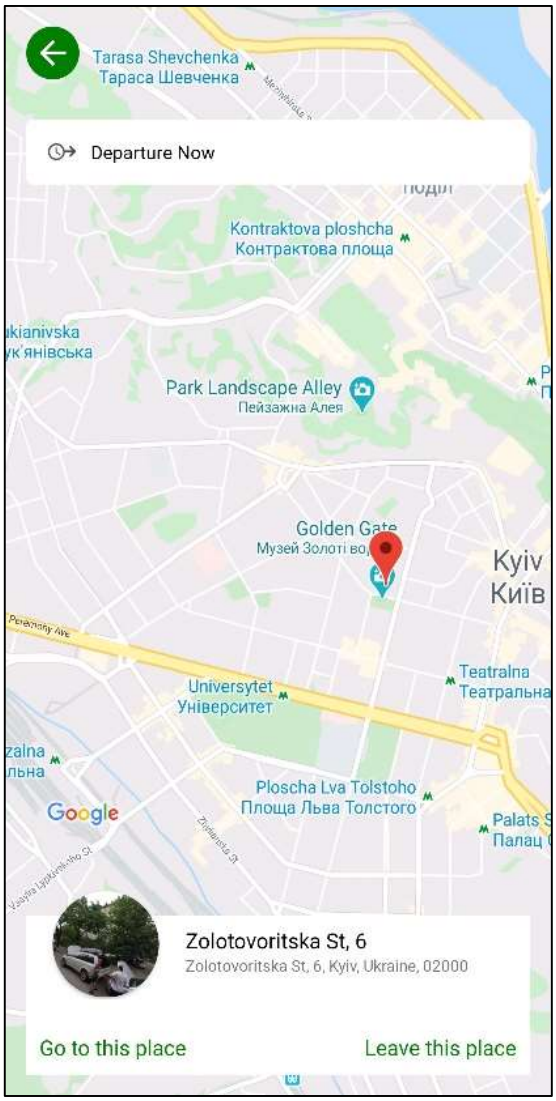


Рис. 3.15 Точка на карті

Якщо користувач на головному екрані натисне на будь-яке місце на карті, він потрапить на екран деталей цього місця (Рис. 3.1). З цього екрану користувач може указати, що це місце належатиме до його маршруту, та якою точкою ця точка там буде (початком чи кінцем). Одразу після вибору типу точки, користувач перейде на екран вибору іншої точки (Рис. 3.2).

3.12. Календар

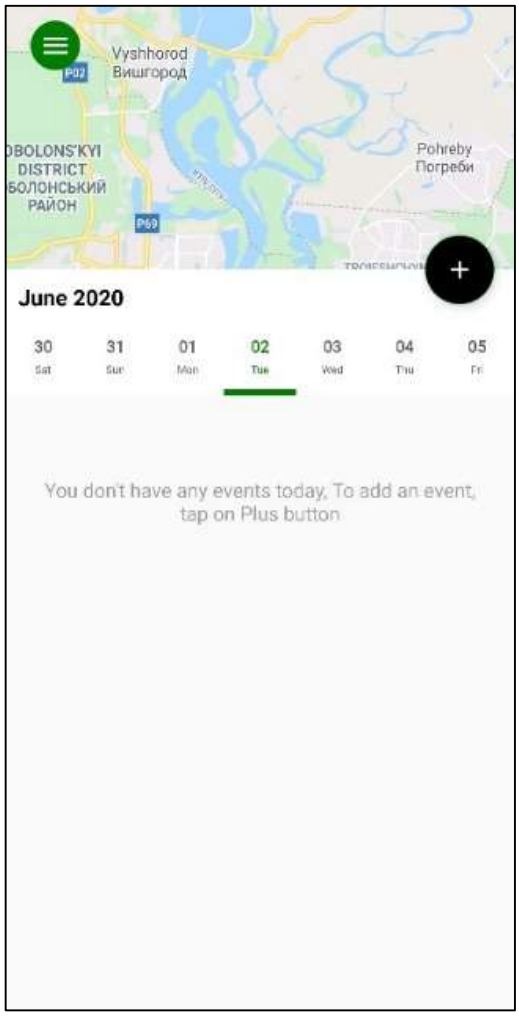


Рис. 3.16 Календар

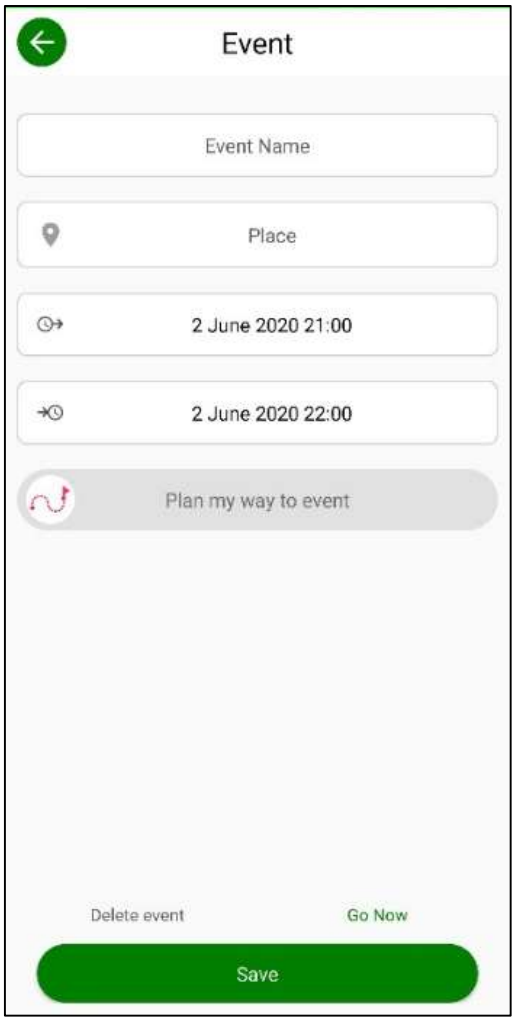


Рис. 3.17 Створення події

Ще одним екраном додатку є екран подій. Він складається з двох частин. Екран перегляду подій на тиждень (Рис. 3.16) та екран створення, редагування та перегляду події (Рис. 3.17). Якщо користувач хоче створити подію, він на екрані календаря натискає на кнопку зі знаком “+”. Якщо користувач хоче переглянути чи редагувати подію, він натискає на саму подію в календарі. В календар також підтягуються події з календаря андроїд.

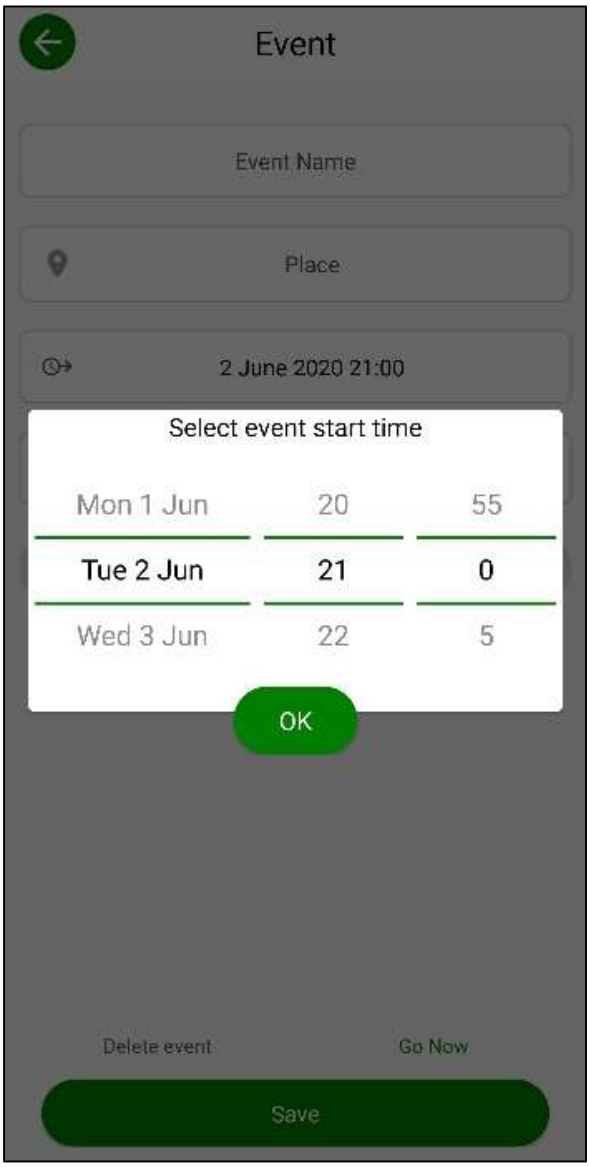


Рис. 3.18. Вибір часу події

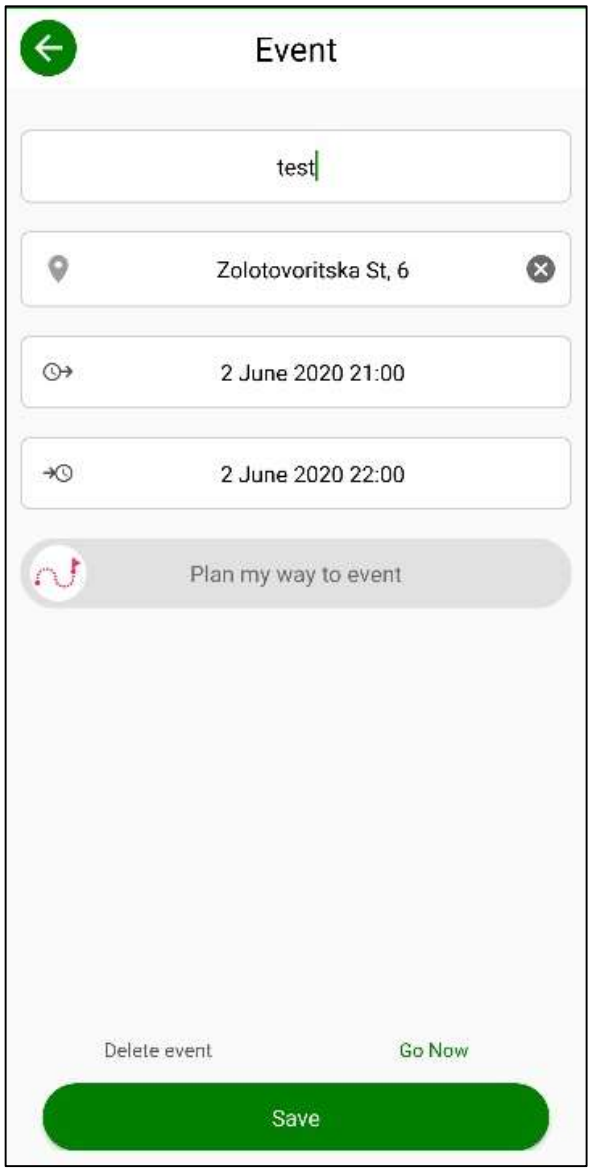


Рис. 3.19. Вибрана локація для події

При створенні чи редагуванні події користувачу запропонують заповнити кілька полів:

- ім'я події,
- місце події,
- час початку,
- час кінця,
- маршрут до місця події.

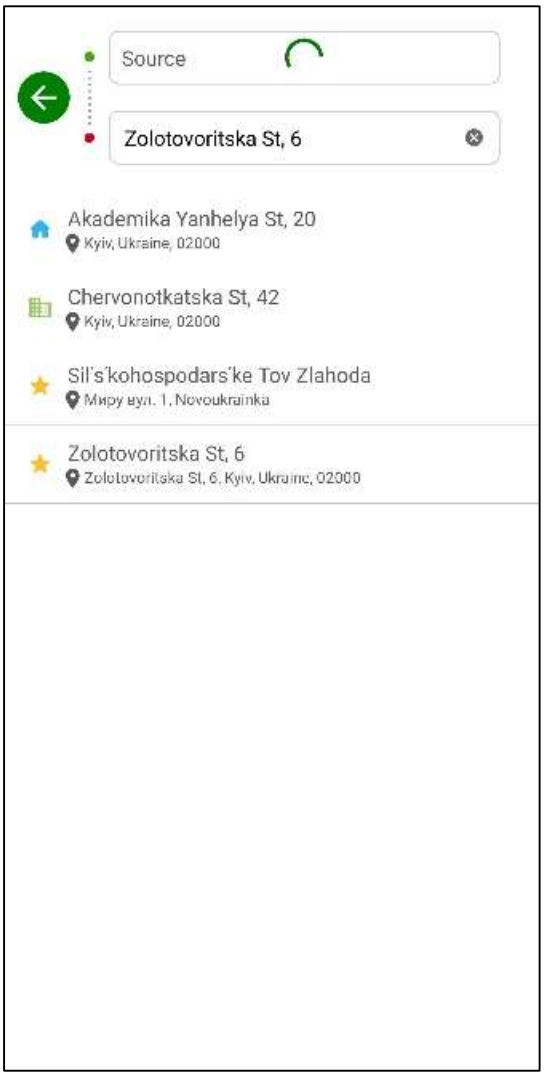


Рис. 3.20. Точок маршруту для події

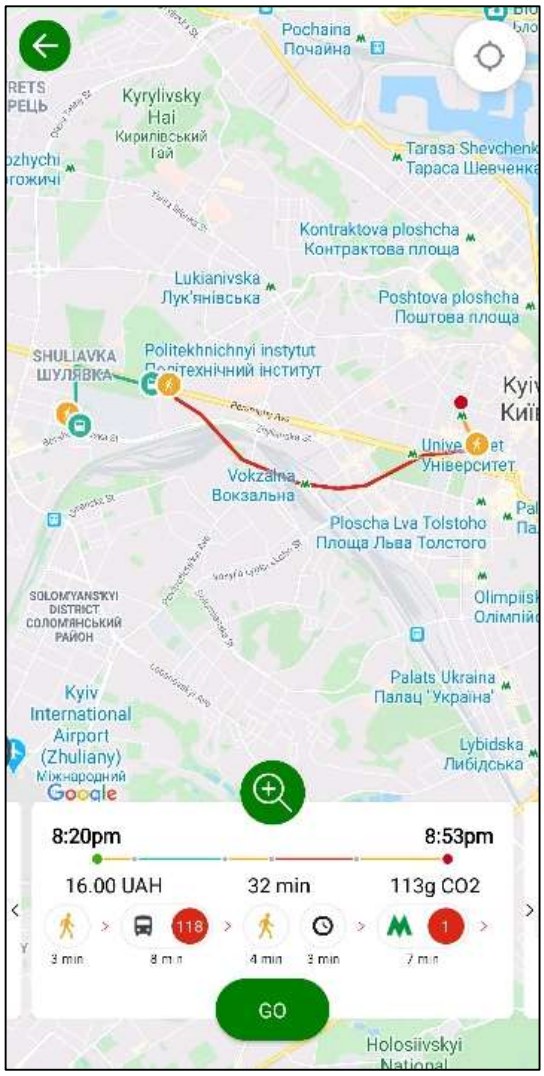


Рис. 3.21. Точок маршруту для події

Після натиснення на кнопку додавання маршруту користувач перейде на екран вибору точок маршруту, де місце події буде зазначене як кінцева точка. А час початку події буде зазначено як час прибуття.

Після вибору точок та часу користувач перейде на екран вибору маршруту. Після його вибору користувач не перейде до навігації як це було раніше, а повернеться до екрану редагування події, попередньо додаток зробить знімок маршруту та додасть його до події. Знімок потрібен, щоб показати його в сповіщенні про подію.

Рис. 3.22. Подія з маршрутом

Рис. 3.23. Готова подія в календарі

Після додавання маршруту подія виглядає так (Рис. 3.22.). Якщо користувач хоче відредагувати маршрут, йому достатньо просто клацнути на нього.

Після створення події вона буде відображатися на екрані події (Рис. 3.23.).

3.13. Метро

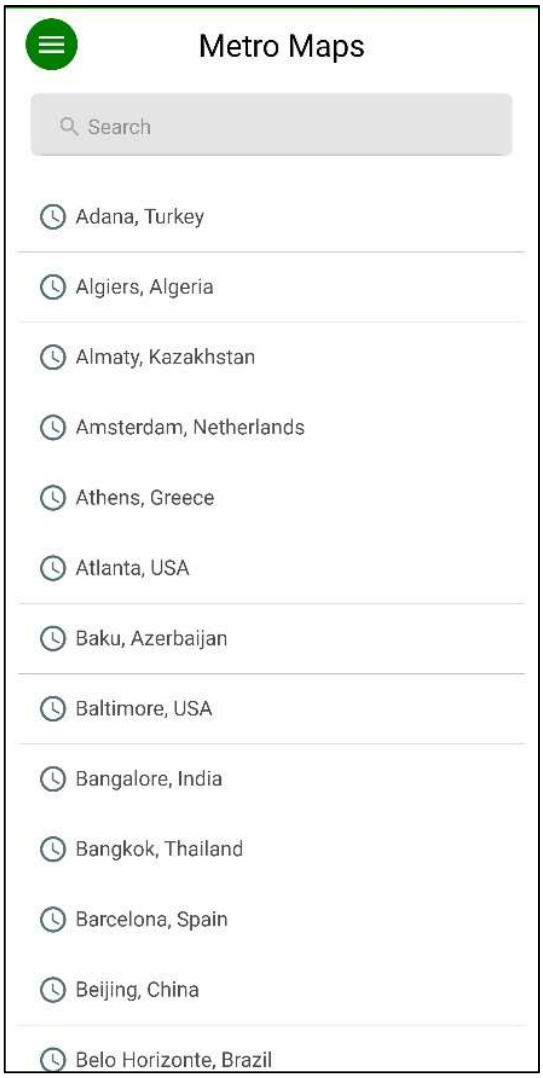


Рис. 3.24. Список метро

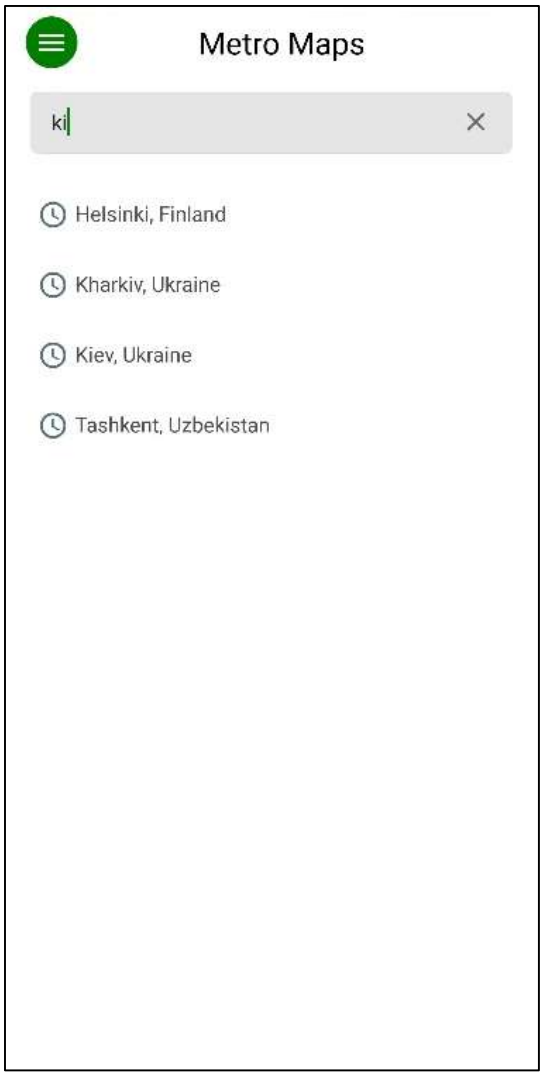


Рис. 3.25. Пошук метро

В додатку зібрані більшість мап метро світу. Якщо користувач хоче знайти там своє місто, йому варто просто почати вводити його в пошуковий рядок. Після знаходження потрібного міста його можна переглянути.

Рис. 3.26. Перегляд міста

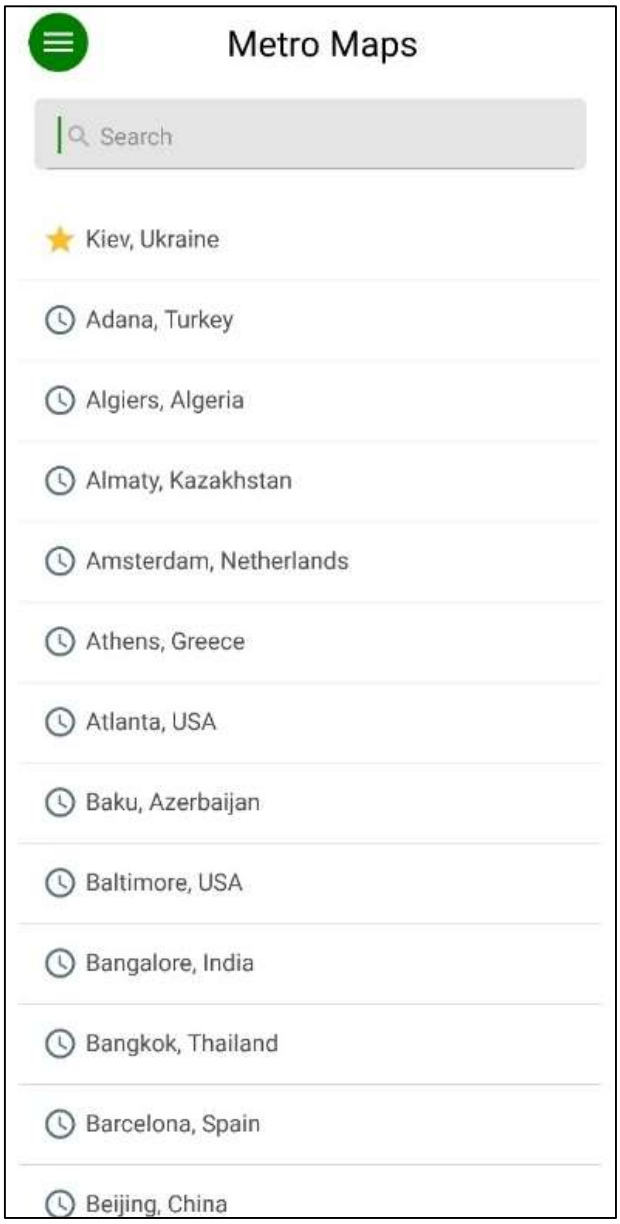


Рис. 3.27. Список метро

Після натиснення на потрібне місто, користувач потрапляє на екран перегляду станцій метро. На екрані реалізовані функції галереї (користувач може наближати та віддаляти вигляд).

Після перегляду, коли користувач повернеться, метро буде позначено як улюблене, і більше не потрібно буде його шукати.

ВИСНОВКИ ДО РОЗДІЛУ 3

В даному розділу було детально розібрано кожен екран застосунку. Також було розглянуто, за що відповідають кнопки на кожному екрані, та описано. що відбудеться після натиснення кожної з них. Також було показано, які можливості має додаток на своєму фінальному етапі.

ВИСНОВОК

В роботі було поставлено завдання розробити зручний навігаційний додаток для системи Android. Розглянемо результати роботи.

- Було проаналізовано декілька провідних додатків та переглянуті їх можливості. На основі даних отриманих в ході аналізу було поставлено завдання роботи.
- Розглянуто архітектурні патерни та фреймворки. Детально описана їх робота та описано як ними користуватися.
- Був розроблений додаток на основі зібраних даних, який має свої унікальні риси, такі як:
    - зручний та інтуїтивний дизайн,
    - планування поїздки в майбутньому та нагадування про неї,
    - редагування запропонованого маршруту,
    - автоматичне визначення місця дому та роботи шляхом рейтингової системи.

Опираючись на ці факти, можна вважати програмний продукт готовим, а поставлене завдання виконаним.

СПИСОК ДЖЕРЕЛ

1. Google Maps [Електронний ресурс] – Режим доступу до ресурсу: https://play.google.com/store/apps/details?id=com.google.android.apps.maps&hl=en.
2. MAPS.ME [Електронний ресурс] – Режим доступу до ресурсу: https://play.google.com/store/apps/details?id=com.mapswithme.maps.pro&hl=en.
3. WAZE [Електронний ресурс] – Режим доступу до ресурсу: https://play.google.com/store/apps/details?id=com.waze&hl=en.
4. Getting Started with MVP [Електронний ресурс].– Режим доступу до ресурсу: https://www.raywenderlich.com/7026-getting-started-with-mvp-model-view-presenter-on-android.
5. Glide Tutorial for Android [Електронний ресурс]. – Режим доступу до ресурсу: https://www.raywenderlich.com/2945946-glide-tutorial-for-android-getting-started.
6. Using Retrofit 2.x as REST client [Електронний ресурс]. – Режим доступу до ресурсу: https://www.vogella.com/tutorials/Retrofit/article.html.
7. Using RxJava 2 [Електронний ресурс]. – Режим доступу до ресурсу: https://www.vogella.com/tutorials/RxJava/article.html.
8. RxJava — Schedulers — What, when and how to use it? [Електронний ресурс]. – Режим доступу до ресурсу: https://android.jlelse.eu/rxjava-schedulers-what-when-and-how-to-use-it-6cfc27293add.
9. Observable [Електронний ресурс] – Режим доступу до ресурсу: http://reactivex.io/rxjs/class/es6/Observable.js~Observable.html.
10. Places API for Android [Електронний ресурс]: developers.google.com – Режим доступу: https://developers.google.com/places/android-api/start.
11. Роджерс Р., Ломбардо Д. Android. Разработка приложений [Текст] / Роджерс Р., Ломбардо Д. – М.: ЭКОМ Паблишерз, 2010. — 400 с.

12. Этапы разработки [Електронний ресурс]: itech-mobile.ru – Режим доступу: http://itech-mobile.ru/stages.html.
13. Родной язык Андроида [Електронний ресурс]: toster.ru – Режим доступу: https://toster.ru/q/8860.
14. Directions API [Електронний ресурс]: developers.google.com – Режим доступу: https://developers.google.com/maps/documentation/directions/start.

ДОДАТОК 1

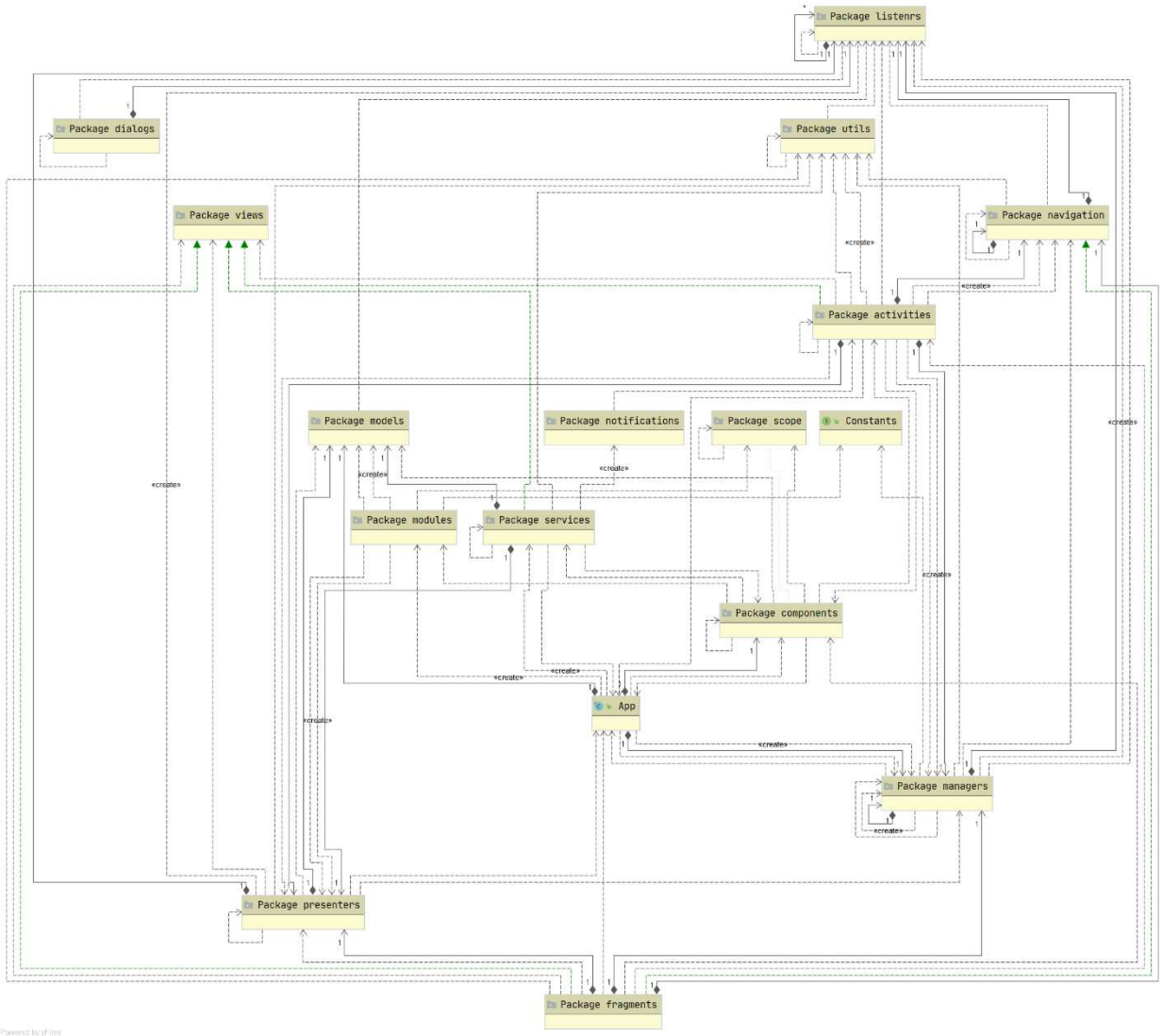
Додаток побудови маршрутів

Діаграма класів. Структурна схема.

ІАЛЦ.467100.004 Д1

Листів 1

2020



					ІАЛЦ.467100.004 Д1	
Змін.	*Арк.*	*№ докум.*	*Підпис*	*Дата*		
Розробив		*Ішенко Б. А.*			*Додаток побудови маршрутів Додаток 1*	*Літ.* / *Аркуш* 1 / *Аркушів* 1
Перевір.		*Алещенко О.В.*				*КПІ ім. Ігоря Сікорського ФІОТ ІО-63*
Н. контр.		*Симоненко В.П.*				
Затверд.		*Стиренко С.Г.*				

ДОДАТОК 2

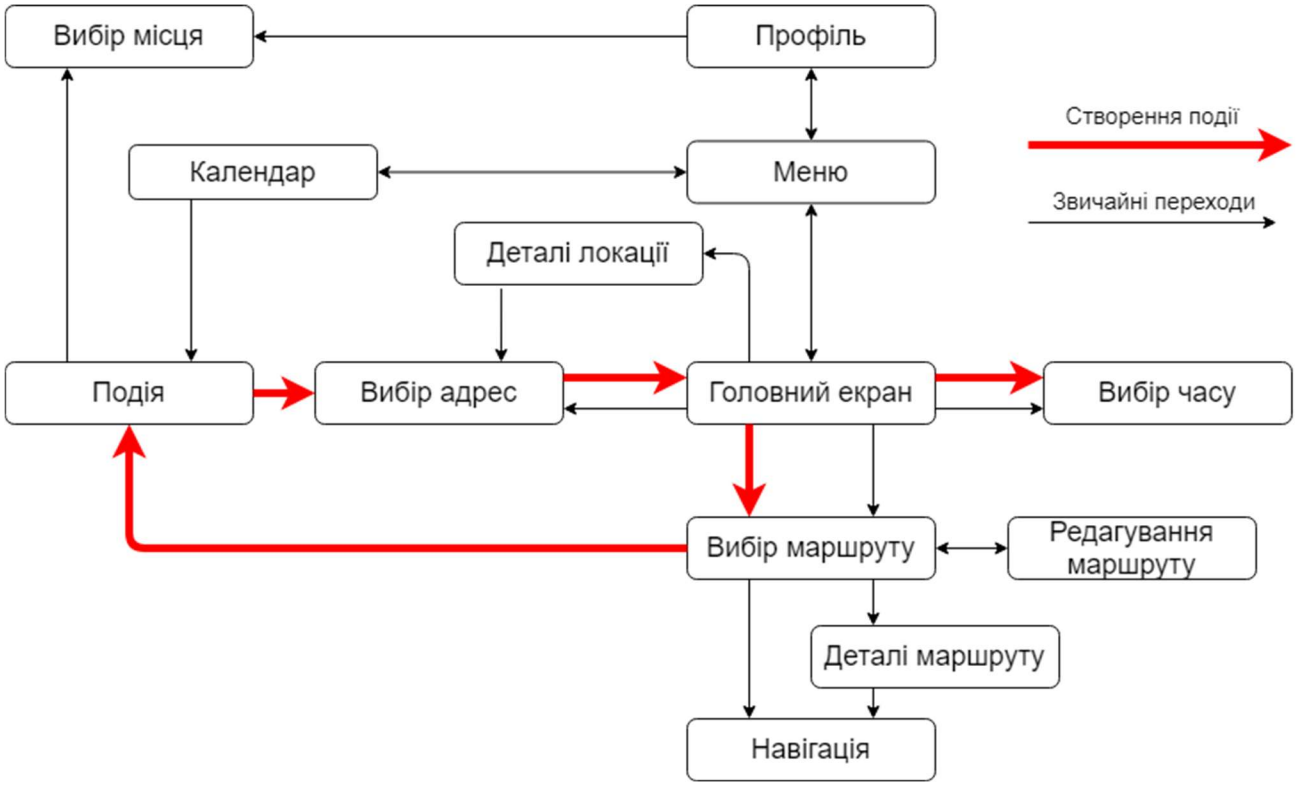
Додаток побудови маршрутів

Діаграма навігації по додатку. Схема взаємодії

ІАЛЦ.467100.005 Д2

Листів 1

2020



					ІАЛЦ.467100.005 Д2	
Змін.	Арк.	№ докум.	Підпис	Дата		
Розробив		Ішенко Б. А.			Додаток побудови маршрутів Додаток 2	Літ. / Аркуш / Аркушів 1 / 1
Перевір.		Алещенко О.В.				КПІ ім. Ігоря Сікорського ФІОТ ІО-63
Н. контр.		Симоненко В.П.				
Затверд.		Стиренко С.Г.				

ДОДАТОК 3

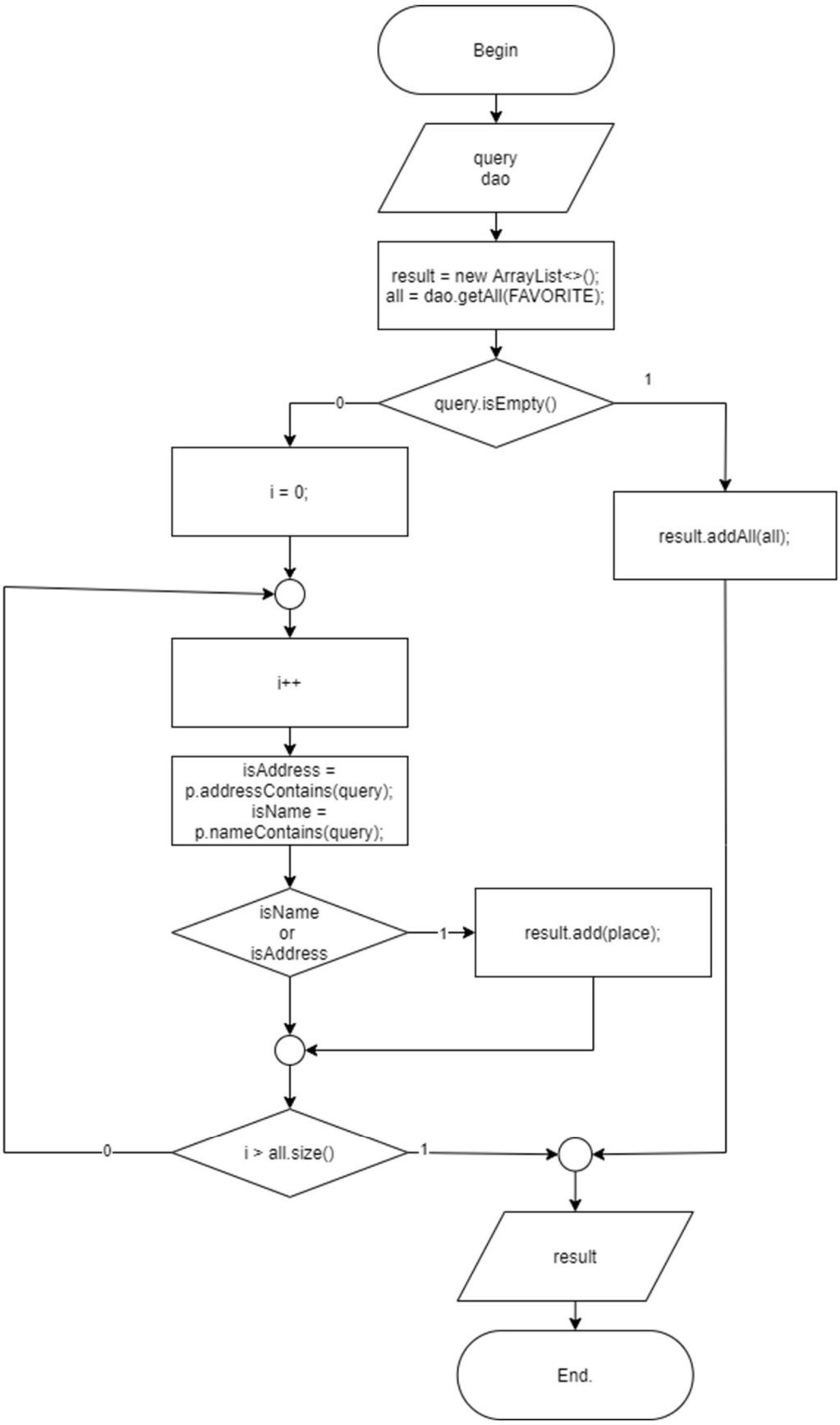
Додаток побудови маршрутів

Алгоритм заміни сегменту маршруту. Функціональна схема

ІАЛЦ.467100.06 Д3

Листів 1

2020



					ІАЛЦ.467100.006 Д3	
Змін.	*Арк.*	*№ докум.*	*Підпис*	*Дата*		
Розробив		*Ішенко Б. А.*			*Додаток побудови маршрутів* *Додаток 3*	*Літ.* / *Аркуш* / *Аркушів*: 1 / 1
Перевір.		*Алещенко О.В.*				*КПІ ім. Ігоря Сікорського* *ФІОТ ІО-63*
Н. контр.		*Симоненко В.П.*				
Затверд.		*Стиренко С.Г.*				

ДОДАТОК 4

Додаток побудови маршрутів

Лістинг програми

ІАЛЦ.467100.07 Д4

Листів 12

2020

```
public final class MainActivity
        extends BaseActivity<MainView>
        implements MainView {

    public static final int SHOW_ROUTE_REQUEST = 2002;

    @BindView(R.id.navigation_view)
    protected NavigationView navigationView;
    @BindView(R.id.drawer_layout)
    protected DrawerLayout drawerLayout;

    protected Fragment mapFragment;

    @Inject
    protected MainPresenter presenter;

    public static Navigator navigator;
    private int requestCode;

    public static void launch(Context context) {
        context.startActivity(new Intent(context, MainActivity.class));
    }

    private static long lastUserExitTime;

    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main);
        ButterKnife.bind(this);

        getPresentersComponent().inject(this);
        registerPresenterLifecycle(presenter, this);

        mapFragment = new GoogleMapFragment();
        FragmentTransaction transaction = getSupportFragmentManager().beginTransaction();
        transaction.replace(R.id.map_container, mapFragment);
        transaction.addToBackStack(null);
        transaction.commit();

        navigator = new Navigator(savedInstanceState,
                                  getSupportFragmentManager(),
                                  this::onFragmentChanged,
                                  this::onRootChanged,
                                  this::onMenuClick);

        obtainIntent();
        initLeftSideMenu();
    }

    private void onFragmentChanged(NavigableFragment navigableFragment) {}

    private void onRootChanged(NavigableFragment navigableFragment) {}

    private void initLeftSideMenu() {
        navigationView.setNavigationItemSelectedListener(this::onMenuItemClickListener);
        navigationView.setItemIconTintList(null);
    }

    private void obtainIntent() {
```

```
    if (requestCode == SHOW_ROUTE_REQUEST) {
        RouteShowingManager.show(presenter.getStateData(),
                                 RouteItem.fromGson(getIntent().getStringExtra(
                                       getString(R.string.extra_route_item))),
                                 navigator,
                                 this);
        requestCode = 0;
    }
}

private boolean onMenuItemClickListener(MenuItem item) {
    drawerLayout.closeDrawer(GravityCompat.START);
    final StateData stateData = presenter.getStateData();
    stateData.setAgendaWaitingForRoute(false);
    requestCode = 0;
    switch (item.getItemId()) {
        case R.id.nav_agenda:
            stateData.clearRoteData();
            navigator.replace(Screen.AGENDA);
            break;
        case R.id.nav_navigation:
            navigator.replace(Screen.MAIN_NAVIGATION);
            break;
        case R.id.nav_metro:
            stateData.clearRoteData();
            navigator.replace(Screen.METRO);
            break;
        case R.id.nav_profile:
            stateData.clearRoteData();
            ProfileActivity.launch(this);
            break;
    }
    return true;
}

private void onMenuClick() {
    if (!drawerLayout.isDrawerOpen(GravityCompat.START)) {
        drawerLayout.openDrawer(GravityCompat.START);
    }
}

@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    navigator.onSaveInstanceState(outState);
}

@Override
public void onBackPressed() {
    navigator.getCurrentFragment().onBackPressed();
}

@Override
public void showLoginSuccess() {
    ((Fragment) navigator.getCurrentFragment()).onResume();
}

@Override
protected void onPause() {
    super.onPause();
    lastUserExitTime = System.currentTimeMillis();
```

```
    }

    @Override
    protected void onResume() {
        super.onResume();
        long now = System.currentTimeMillis();
        long minute = MINUTES.toMillis(1);
        long dif = now - lastUserExitTime;
        if (dif > minute) {
            presenter.getStateData().clearRoteData();
        }
    }
}
public final class MainNavigationFragment
        extends BaseFragment<DirectionsView>
        implements DirectionsView, AgendaView {

    private static final int GET_FAVORITE_PLACE_REQUEST_CODE = 4868;

    @BindView(R.id.text_time)
    protected TextView textTime;
    @BindView(R.id.address_path_input)
    protected TextView addressPathInput;

    @Inject
    protected MainNavigationPresenter presenter;
    @Inject
    protected AgendaPresenter agendaPresenter;

    @Override
    public View onCreateView(@NonNull LayoutInflater inflater,
                             @Nullable ViewGroup container,
                             @Nullable Bundle savedInstanceState) {
        return inflater.inflate(R.layout.fragment_main_navigation, container, false);
    }

    @Override
    public void onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {
        super.onViewCreated(view, savedInstanceState);
        ButterKnife.bind(this, view);

        getPresentersComponent().inject(this);
        registerPresenterLifecycle(presenter, this);

        mapManager.setOnMapClickListener(presenter::getPlace);

    }

    @Override
    public void onResume() {
        super.onResume();
        if ((agendaPresenter != null) && !agendaPresenter.isBind()) {
            agendaPresenter.bindView(this);
        }

        presenter.updateDirectionRequest();

        mapManager.setOnMapClickListener(presenter::getPlace);

        presenter.getDirectionsSilently();
    }
```

```
@Override
public void onPause() {
   super.onPause();
   if ((agendaPresenter != null) && agendaPresenter.isBind()) {
      agendaPresenter.unbindView();
   }
   mapManager.setOnMapClickListener(null);
}

@Override
public void showPlace(Place place) {
   presenter.getStateData().setSelectedPlace(place);
   MainActivity.navigator.add(PLACE_DETAILS);
}

@OnClick(R.id.action_toggle_menu)
protected void onMenu() {
   navigator.showMenu();
}

@OnClick(R.id.text_time)
protected void onDatePicker() {
   navigator.add(Screen.DATE_PICKER);
}

@OnClick(R.id.address_path_input)
protected void onAddressPath() {
   navigator.add(ADDRESS_PICKER);
}

@Override
public void showAddress(String text) {
   addressPathInput.setText(text);
}

@Override
public void showTime(TimePickerEntity timePickerEntity) {
   textTime.setCompoundDrawablesWithIntrinsicBounds(timePickerEntity.getDrawable(), 0, 0, 0);
   textTime.setText(timePickerEntity.getText(getContext()));
}

@Override
public void onRequestPermissionsResult(int requestCode,
                                       @NonNull String[] permissions,
                                       @NonNull int[] grantResults) {
   if (PermissionUtils.hasSelfPermissions(getContext(), new String[]{
         PermissionUtils.READ_CALENDAR, PermissionUtils.WRITE_CALENDAR})) {
      agendaPresenter.bindView(this);
      agendaPresenter.allowCalendar();
      final Calendar calendar = Calendar.getInstance();
      DateUtils.roundCalendarToDays(calendar);
      agendaPresenter.getEvents(calendar, Calendar.getInstance());
   }
   super.onRequestPermissionsResult(requestCode, permissions, grantResults);
}


@OnClick(R.id.fabAgenda)
protected void onAgenda() {
   agendaPresenter.setCalendarAllowed(PermissionUtils.requestCalendarPermissions(this));
```

```
    final Calendar calendar = Calendar.getInstance();
    DateUtils.roundCalendarToDays(calendar);
    agendaPresenter.getEvents(calendar, Calendar.getInstance());
}

@OnClick(R.id.fabGps)
protected void onGps() { mapManager.animateToMe(); }

@Override
public void showEvents(List<Event> events) {
    presenter.processEvents(events);
}

@Override
public void showEndLocation(Location endLocation) {
    presenter.getStateData().getRoutePoints().setDestination(new Place(endLocation));
    navigator.add(ADDRESS_PICKER);
}

@Override
public void showNoEvent() {
    AndroidUtils.showLongToast(getContext(), R.string.no_events_with_route_for_current_day);
}

@OnClick(R.id.fabHome)
protected void onHome() {
    presenter.getHomePlace();
}

@Override
public void showHomeAddressPicker(Place homePlace) {
    if (homePlace == null || homePlace == Place.NULL) return;
    presenter.getStateData().getRoutePoints().setDestination(homePlace);
    navigator.add(ADDRESS_PICKER);
}

@OnClick(R.id.fabBusiness)
protected void onWork() {
    presenter.getWorkPlace();
}

@Override
public void showWorkAddressPicker(Place workPlace) {
    if (workPlace == Place.NULL || workPlace == null) return;
    presenter.getStateData().getRoutePoints().setDestination(workPlace);
    navigator.add(ADDRESS_PICKER);
}

@OnClick(R.id.fabFavorite)
protected void onFavorite() {
    presenter.getTheFavoritesPlace();
}

@Override
public void showFavorite(Place place) {
    if (place == Place.NULL) {
        AndroidUtils.showLongToast(getContext(), R.string.no_favorite_place);
    } else {
        presenter.getStateData().getRoutePoints().setDestination(place);
        navigator.add(ADDRESS_PICKER);
    }
```

```
    }

    @OnClick(R.id.action_go)
    protected void onClickGo() {
        presenter.initRoute(true);
    }

    @Override
    public void animateTo(LatLng source, LatLng destination) {
        mapManager.moveTo(Arrays.asList(source, destination), false);
    }

    @Override
    public void showLoading(boolean isLoading) {
        LoadingDialog.set(isLoading, getChildFragmentManager());
    }

    @Override
    public void onActivityResult(int requestCode, int resultCode, Intent data) {
        super.onActivityResult(requestCode, resultCode, data);
        if (resultCode != RESULT_OK) return;

        if (requestCode == GET_FAVORITE_PLACE_REQUEST_CODE) {
            RoutePoints routePoints = presenter.getStateData().getRoutePoints();
            routePoints.setDestination(data.getParcelableExtra(getString(R.string.extra_place)));
            navigator.add(ADDRESS_PICKER);
        }
    }

    @Override
    public void showDirections(List<RouteItem> items) {
        presenter.getStateData().setRouteItems(items);

        if (presenter.getStateData().getRouteItems() == null) {
            AndroidUtils.showLongToast(getContext(), R.string.data_is_not_ready_yet);
        } else {
            navigator.add(SELECT_ROUTE);
        }
    }

    @Override
    public void showError() {
        showLoading(false);
    }

    @Override
    public void onBackPressed() {
        FragmentActivity activity = getActivity();
        if (activity == null) {
            System.exit(0);
        } else {
            activity.finish();
        }
    }
}
public final class MainNavigationPresenter
        extends BasePresenter<DirectionsView> {

    private final DirectionsDataSource directionsDataSource;
    private final PreviousSessionDataSource previousSessionDataSource;
    private final PlaceDataSource placeDataSource;
```

```
    private final PlacesDataSource placesDataSource;

    private DirectionRequestData requestData;
    private DirectionRequestData justRequestedData;
    private Disposable currentDirectionDisposable;
    private Disposable currentSpecialAbilitiesDisposable;
    private ObservableField<Pair<DirectionRequestData, List<RouteItem>>> currentResponse = new
ObservableField<>();
    private Consumer<Pair<DirectionRequestData, List<RouteItem>>> responseCallBack = pair -> {
        if (pair == null) return;
        if (view == null) {
            viewObservable.addWeakCallbackOnUpdate(view -> {
                view.showLoading(false);
                view.showDirections(pair.second);
            });
        } else {
            view.showLoading(false);
            view.showDirections(pair.second);
        }
    };

    public MainNavigationPresenter(StateData stateData,
                                   DirectionsDataSource directionsDataSource,
                                   PreviousSessionDataSource previousSessionDataSource,
                                   PlaceDataSource placeDataSource,
                                   PlacesDataSource placesDataSource) {
        super(stateData);
        this.directionsDataSource = directionsDataSource;
        this.previousSessionDataSource = previousSessionDataSource;
        this.placeDataSource = placeDataSource;
        this.placesDataSource = placesDataSource;
    }

    public void initRoute(boolean initSpecialAbilities) {
        if (!DirectionRequestData.isValid(requestData)) {
            view.showLoading(false);
            return;
        }
        view.showLoading(true);

        boolean needNewRequest = true;

        if (requestData.equals(justRequestedData)) {
            if (currentDirectionDisposable.isDisposed()) {
                final Pair<DirectionRequestData, List<RouteItem>> entry = currentResponse.get();
                if (entry != null) {
                    if (entry.first.equals(requestData)) {
                        if (entry.second != null) {
                            responseCallBack.accept(entry);
                            needNewRequest = false;
                        }
                    }
                }
            } else {
                view.animateTo(requestData.getRoutePoints().getSource().getLatLng(),
                        requestData.getRoutePoints().getDestination().getLatLng());
                currentResponse.addWeakCallbackOnUpdate(responseCallBack);
                needNewRequest = false;
            }
        }
        if (needNewRequest) {
```

```
        currentResponse.set(null);
        currentDirectionDisposable = getDirectionsDisposable(initSpecialAbilities);
        currentResponse.addWeakCallbackOnUpdate(responseCallBack);
    }
}

@NotNull
private Disposable getDirectionsDisposable(boolean initSpecialAbilities) {
    disposeCurrentRequest();
    return directionsDataSource
            .getDirections(requestData)
            .compose(new AsyncTransformer<>())
            .doOnSubscribe(disposable -> justRequestedData = requestData)
            .subscribe(items -> {
                        currentResponse.set(new Pair<>(requestData, items));
                        if (initSpecialAbilities) {
                            currentSpecialAbilitiesDisposable = getSpecialAbilitiesDisposable(items);
                        }
                    },
                    new RxErrorAction(view, view::showError));
}

private Disposable getSpecialAbilitiesDisposable(List<RouteItem> items) {
    return directionsDataSource.fillWithSpecialAbilities(items)
                        .compose(new AsyncTransformer<>())
                        .doOnNext(stateData::setRouteItems)
                        .subscribe(routeItems -> {}, Throwable::printStackTrace);
}

public void getDirectionsSilently() {
    if (!DirectionRequestData.isValid(requestData)) return;
    if (requestData.equals(justRequestedData)) return;
    currentDirectionDisposable = getDirectionsDisposable(true);
}

public void initDirectionRequest(RouteSegmentManager manager, long departureTime) {
    requestData = new DirectionRequestData(manager.getRoutPointsFromSelectedToEnd(),
                                manager.getDepartureMillisFromSelected(departureTime),
                                ARRIVAL);
}

public void updateDirectionRequest() {
    requestData = stateData.getRequestData();
    if (requestData.isRoutePointsValid()) {
        view.showAddress(requestData.getRoutePoints().getDestination().getName());
    }
    view.showTime(requestData.getTimePickerEntity());
}

private void disposeCurrentRequest() {
    if (currentDirectionDisposable != null && !currentDirectionDisposable.isDisposed()) {
        currentDirectionDisposable.dispose();
    }
    if (currentSpecialAbilitiesDisposable != null && !currentSpecialAbilitiesDisposable.isDisposed()) {
        currentSpecialAbilitiesDisposable.dispose();
    }
}

public void getPlaceDescription(LatLng latLng) {
    addDisposable(
            placeDataSource
```

```
                .getPlaceDescription(latLng.latitude, latLng.longitude)
                .map(description -> mapDescription(latLng, description))
                .compose(new AsyncTransformer<>())
                .subscribe(view::showPlace, new RxErrorAction(view)));
    }

    public void getPlace(LatLng latLng) {
        if (view == null) return;
        addDisposable(
                placeDataSource
                .getPlaceDescription(latLng.latitude, latLng.longitude)
                .map(description -> mapDescription(latLng, description))
                .compose(new AsyncTransformer<>())
                .doOnSubscribe(disposable -> view.showLoading(true))
                .doAfterTerminate(() -> view.showLoading(false))
                .subscribe(view::showPlace, new RxErrorAction(view)));
    }

    public void getHomePlace() {
        addDisposable(placesDataSource.getHome()
                        .compose(new AsyncTransformer<>())
                        .subscribe(view::showHomeAddressPicker,
                                new RxErrorAction(view)));
    }

    public void getWorkPlace() {
        addDisposable(placesDataSource.getWork()
                        .compose(new AsyncTransformer<>())
                        .subscribe(view::showWorkAddressPicker,
                                new RxErrorAction(view)));
    }

    public void getTheFavoritesPlace() {
        addDisposable(
                Observable.zip(placesDataSource.getAllFavorites(null),
                        placesDataSource.getHome(),
                        placesDataSource.getWork(),
                        (places, home, work) -> {
                            for (Place place : places) {
                                if (!place.equals(home) && !place.equals(work)) return place;
                            }
                            return Place.NULL;
                        }
                       )
                .compose(new AsyncTransformer<>())
                .subscribe(view::showFavorite, new RxErrorAction(view)));
    }

    public void processEvents(List<Event> events) {
        if (events == null || events.isEmpty()) {
            view.showNoEvent();
            return;
        }
        events = filter(events,
                event -> event.getRouteItem() != null && !event.getRouteItem().isEmpty());
        final Event event = max(events, (o1, o2) -> o1.getStartDate().compareTo(o2.getStartDate()));
        if (event == null) {
            view.showNoEvent();
            return;
        }
```

```
        try {
            view.showEndLocation(RouteItem.fromGson(event.getRouteItem())
                                 .getFirstLeg()
                                 .getEndLocation());
        } catch (com.google.gson.JsonSyntaxException e) {
            e.printStackTrace();
            view.showNoEvent();
        }
    }
}
public class MapboxConverterManager {

    public static void build(Context context,
                             Step step,
                             Consumer<DirectionsRoute> onSuccess,
                             Consumer<Throwable> onError) {

        final Location startLocation = step.getStartLocation();
        final Location endLocation = step.getEndLocation();
        NavigationRoute.Builder builder = NavigationRoute
                .builder(context)
                .accessToken(Mapbox.getAccessToken())
                .profile(step.getTravelMode().getNavigationProfile())
                .origin(Point.fromLngLat(startLocation.getLng(), startLocation.getLat()))
                .destination(Point.fromLngLat(endLocation.getLng(), endLocation.getLat()));

        Deviation deviation = step.getDeviation();
        if (deviation != null && deviation.getDeviationPoints() != null) {
            List<DeviationPoint> points = deviation.getDeviationPoints();
            for (DeviationPoint point : points) {
                Location location = point.getLocation();
                builder.addWaypoint(Point.fromLngLat(location.getLng(), location.getLat()));
            }
            if (step.getTravelMode() == TravelMode.DRIVING && points.size() > 1) { // if there are more then one point,
we need to use profile without traffic
                builder.profile(DirectionsCriteria.PROFILE_DRIVING);
            }
        }

        builder.build()
                .getRoute(new Callback<DirectionsResponse>() {
                    @Override
                    public void onResponse(Call<DirectionsResponse> call,
                                           Response<DirectionsResponse> response) {
                        if (response.body() == null || response.body().routes().size() < 1) {
                            onError.accept(null);
                        } else {
                            DirectionsRoute route = response.body().routes().get(0);
                            onSuccess.accept(route);
                        }
                    }

                    @Override
                    public void onFailure(Call<DirectionsResponse> call, Throwable t) {
                        onError.accept(t);
                    }
                });
    }
}
public final class AlarmHelper {
```

```
private final AlarmManager alarmManager;
@NonNull
private final Context context;

public AlarmHelper(Context context) {
   this.alarmManager = (AlarmManager) context.getSystemService(ALARM_SERVICE);
   this.context = context;
}

public void removeEvent(Event currentEvent) {
   Intent intent = new Intent();
   PendingIntent pendingIntent = PendingIntent.getBroadcast(context.getApplicationContext(),
                                          (int) currentEvent.getId(),
                                          intent,
                                          0);
   alarmManager.cancel(pendingIntent);
}

public void setNextEvent(Event event) {
   final Intent intent = new Intent(context, IntentReceiver.class);
   intent.setAction(IntentReceiver.EVENT_ACTION);
   intent.putExtra(context.getString(R.string.extra_event_id), event.getId());

   final PendingIntent alarmIntent = getBroadcast(
         context,
         IntentReceiver.EVENT_REQUEST_CODE,
         intent,
         PendingIntent.FLAG_UPDATE_CURRENT
                                    );

   setAlarm(event.getStartDate().getTimeInMillis() - TimeUnit.MINUTES.toMillis(5),
          alarmIntent);
}

private void setAlarm(long time, PendingIntent alarmIntent) {
   try {
      if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
         alarmManager.setExactAndAllowWhileIdle(AlarmManager.RTC_WAKEUP, time, alarmIntent);
      } else if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT) {
         alarmManager.setExact(AlarmManager.RTC_WAKEUP, time, alarmIntent);
      } else {
         alarmManager.set(AlarmManager.RTC_WAKEUP, time, alarmIntent);
      }
   } catch (SecurityException e) {
      e.printStackTrace();
   }
}

public void ensurePlaceDetermineServiceWorking() {
   serRepeatingAlarm(IntentReceiver.COLLECT_HOME_ADDRESS_ACTION,
               IntentReceiver.COLLECT_HOME_ADDRESS_REQUEST_CODE,
               3,
               0);
   serRepeatingAlarm(IntentReceiver.COLLECT_WORK_ADDRESS_ACTION,
               IntentReceiver.COLLECT_WORK_ADDRESS_REQUEST_CODE,
               11,
               30);
   serRepeatingAlarm(IntentReceiver.COLLECT_WORK_ADDRESS_ACTION,
               IntentReceiver.COLLECT_WORK_ADDRESS_REQUEST_CODE_2,
               15,
               30);
```

```
    }

    private void serRepeatingAlarm(String action, int requestCode, int hourOfDay, int minute) {
        final Intent collectHomeIntent = new Intent(context, IntentReceiver.class);
        collectHomeIntent.setAction(action);
        final PendingIntent pendingHomeIntent = PendingIntent.getBroadcast(context,
                                                    requestCode,
                                                    collectHomeIntent,
                                                    PendingIntent.FLAG_UPDATE_CURRENT);

        final Calendar calendarCheckHome = Calendar.getInstance();
        calendarCheckHome.setTimeInMillis(System.currentTimeMillis());
        calendarCheckHome.set(HOUR_OF_DAY, hourOfDay);
        calendarCheckHome.set(MINUTE, minute);
        calendarCheckHome.set(SECOND, 0);
        if (calendarCheckHome.compareTo(Calendar.getInstance()) <= 0) {
            calendarCheckHome.add(DATE, 1);
        }
        alarmManager.setRepeating(RTC,
                          calendarCheckHome.getTimeInMillis(),
                          INTERVAL_DAY,
                          pendingHomeIntent);
    }
}
public class EventAdapter
        extends RecyclerView.Adapter<EventHolder> {

    private List<Event> data;
    private Consumer<Event> onClickListener;

    public EventAdapter(Consumer<Event> onClickListener) {
        this.onClickListener = onClickListener;
        this.data = new ArrayList<>();
    }

    @NonNull
    @Override
    public EventHolder onCreateViewHolder(@NonNull ViewGroup viewGroup, int i) {
        return new EventHolder(LayoutInflater.from(viewGroup.getContext()),
                           viewGroup,
                           position -> onClickListener.accept(data.get(position)));
    }

    @Override
    public void onBindViewHolder(@NonNull EventHolder viewHolder, int position) {
        viewHolder.bind(data.get(position));
    }
    @Override
    public int getItemCount() {
        return data.size();
    }
    public void setEvents(List<Event> events) {
        data.clear();
        data.addAll(events);
        notifyDataSetChanged();
    }
    public void clear() {
        data.clear();
        notifyDataSetChanged();
    }
}
```